APLICACIÓN DE UNA ARQUITECTURA DE PROCESADO DISTRIBUIDO A UNA HERRAMIENTA DE AGREGACIÓN DE COMPONENTES SOFTWARE

Titulación: Ingeniería de Telecomunicaciones

Autor: Sergio Aguilera Cazorla

Director: José Manuel Martín Alonso

Ponente: Marcel Fernández Muñoz

Barcelona, Noviembre 2012

I. AGRADECIMIENTOS

Ante la pregunta “¿Puedes decirme alguna cualidad tuya que consideres notable?” suelo responder “Me gustan los retos complicados” y “Siempre acabo lo que empiezo”. Habitualmente suelen creer que estoy respondiendo con tópicos sacados de un manual: nada más lejos.

En estos momentos tengo la sensación de estar finalizando (“Siempre acabo lo que empiezo”) la tarea más complicada (“Me gustan los retos complicados”) que he emprendido hasta el momento. A una titulación que ya arrastraba la fama de resultar difícil y dura se le ha añadido un proyecto de final de carrera ambicioso, complejo y perteneciente a un campo sobre el cual los ingenieros de telecomunicaciones tradicionalmente intentan pasar de puntillas como es la ingeniería software.

En primer lugar agradecer a mis padres el apoyo y esfuerzo realizado durante los años de estudio para conseguir que pudiese sacar esto adelante. Gracias por el aplomo mostrado cuando las cosas no salían bien y por el regocijo cuando los logros finalmente llegaban. Gracias por apoyar todos los proyectos que he emprendidos y las decisiones que he tomado, así como las que se deberán tomar en años venideros, por irracionales e incomprensibles que vayan a resultar éstas para vosotros.

Debo asimismo agradecer a la Escola Tècnica Superior d’Enginyeria de Telecomunicacions de Barcelona la formación recibida durante todos estos años. Entre los muros del Campus Nord he vivido al mismo tiempo momentos muy buenos y muy duros, y tanto unos como otros me han ayudado a avanzar y crecer profesional y personalmente. Merece una mención especial Marcel Fernández Muñoz por su seguimiento, interés y consejos durante la elaboración de este proyecto. Muchas gracias por el apoyo mostrado desde el primer día, en el cual un estudiante loco irrumpió en tu despacho buscando urgentemente un ponente para un interesante proyecto que le habían propuesto llevar a cabo.

Este proyecto no habría sido posible de no haber tenido la oportunidad de trabajar junto a los grandes profesionales que integran la sección de Comunicaciones y Software de Sener Ingeniería y Sistemas Barcelona. Muchas gracias por el fantástico año y medio que he tenido el placer de compartir con vosotros, que me ha permitido empezar a acumular una valiosa experiencia profesional en un entorno agradable y familiar. Como ya os dije en cierta ocasión, cuando se trabaja al lado de los mejores algo se debe contagiar forzosamente.

En especial, debo agradecer a José Manuel Martín Alonso su sana obsesión en tratar de llevar a cabo fantásticas ideas y proyectos que sirven como potente formación a los jóvenes que estamos empezando. Le agradezco cada consejo y cada minuto que me ha dedicado, y deseo que no pierda nunca esa creatividad de la que, como yo, se podrán beneficiar numerosos compañeros en el futuro. No puedo dejar de mencionar al resto de personas que aportaron su granito de arena en los inicios de este proyecto: Joan Enrich Tudela y, sobretodo, Cruz Herrero Andrés. Vuestro fantástico trabajo e ideas me permiten estar hoy escribiendo esto.

Por último, me veo obligado no a agradecer, sino a enviar un fuerte abrazo, a todas aquellas personas que en algún momento se han sentido partícipes de mi vida, de mis logros y fracasos, de mis celebraciones y tristezas. A los que me conocéis desde hace eones. A los que he conocido durante los últimos años. A los que la vida nos ha llevado finalmente por caminos diferentes, tanto geográficos como psicológicos. A los que la vida aún nos hará caminar por la misma senda algún tiempo más. A los que habéis estado ahí durante la carrera y sobretodo durante un último año y medio en el cual me habría gustado dedicaros muchísimo más tiempo. Debéis saber que al contrario de lo que siempre pueda parecer, vosotros sois la principal (y en ocasiones única) luz que ilumina la salida de cada oscuro túnel en el que decido meterme. Gracias, amigos.

Acaba una etapa muy dura y se inicia otra que no se publicita a sí misma como un camino de rosas.

Challenge Accepted.

II. RESUMEN

En la actualidad, las herramientas software que llevan a cabo un procesado y tratamiento de los datos distribuido y descentralizado resultan esenciales para la buena marcha de multitud de procesos industriales y modelos de negocio. En concreto, las Arquitecturas Orientadas a Servicios (SOA) proponen un escenario en el cual un conjunto de entidades pertenecientes a una red colaboran entre ellas de manera independiente a la tecnología subyacente de cada una de estas entidades, formando de esta manera una federación flexible y auto-gestionada de servicios.

El principal objetivo de este proyecto consiste en dotar de una arquitectura orientada a servicios a una plataforma de edición y ensamblaje de componentes software mediante interfaz gráfica. Para ello se utiliza la plataforma *Jini / Apache River*, un conjunto de librerías y *frameworks* que extienden a la plataforma *Java* para favorecer y facilitar el despliegue de este tipo de arquitecturas.

El punto de partida del proyecto es una plataforma de ensamblaje de componentes software simples escrita en el lenguaje de programación *Java*. Dicha plataforma representa a los componentes software de manera gráfica como cajas negras que es posible interconectar entre sí para conseguir su colaboración mutua, permitiendo de esta manera la construcción de redes de procesado arbitrariamente complejas en base a la agregación, conexionado y encapsulación de multitud de componentes sencillos.

Gracias al entorno propuesto por *Jini / Apache River*, se pretende conseguir que las agregaciones de componentes construidas mediante la herramienta puedan ser publicadas como un servicio. Esto significa que el resto de entidades pertenecientes a la federación pueden hacer un uso remoto de dichos servicios sin conocimiento de su lugar físico de ejecución ni de sus detalles de implementación. Los servicios deben ser reutilizables y deben permitir el uso concurrente por parte de diferentes clientes.

De esta manera, las cajas negras mostradas por la interfaz de usuario de la herramienta de ensamblaje pueden representar procesos software que se ejecutan en la máquina local o en cualquier otro punto de la federación. Es posible la interacción entre ambos tipos de agregaciones de componentes e incluso su reutilización y publicación como un servicio totalmente nuevo. La herramienta de ensamblaje es capaz de visualizar y obtener los servicios disponibles en la federación de la misma manera que visualiza los componentes software propios.

El objetivo secundario de este proyecto es la mejora constante de las funcionalidades ofrecidas por la herramienta de edición de componentes. Además de la adaptación y optimización de la interfaz gráfica para facilitar el uso de las posibilidades de publicación y obtención de servicios, se han implementado multitud de nuevas características que complementan el nuevo proceso de construcción y despliegue de servicios y lo dotan de flexibles y potentes posibilidades.

Al término de este proyecto se dispone de una herramienta versátil capaz de construir redes de procesado complejas como resultado de la agregación de multitud de otras redes y servicios que operan en puntos muy diversos de la federación de entidades *Jini / Apache River*. Es posible asimismo el despliegue e inicialización de dichas redes como servicios públicos en cualquier punto de la federación habilitado a tal efecto, y el control y la gestión remota de los servicios publicados.

III. RESUM

Actualment, les eines software que duen a terme un processat i tractament de les dades distribuït i descentralitzat resulten essencials per a la bona marxa de multitud de processos industrials i models de negoci. En concret, les Arquitectures Orientades a Serveis (SOA) proposen un escenari en el qual un conjunt d'entitats pertanyents a una xarxa col·laboren entre elles de manera independent a la tecnologia subjacent de cadascuna d'aquestes entitats, format d'aquesta manera una federació flexible i autogestionada de serveis.

El principal objectiu d'aquest projecte consisteix a dotar d'una arquitectura orientada a serveis a una plataforma d'edició i assemblatge de components software mitjançant interfície gràfica. Per tal d'aconseguir aquest objectiu s'utilitza la plataforma *Jini / Apache River*, un conjunt de llibreries i *frameworks* que estenen la plataforma *Java* per tal d'afavorir i facilitar el desplegament d'aquest tipus d'arquitectures.

El punt de partida del projecte és una plataforma d'assemblatge de components software simples escrita en el llenguatge de programació *Java*. Aquesta plataforma representa els components software de manera gràfica com a caixes negres que és possible interconnectar entre si per tal d'aconseguir la seva col·laboració mútua, permetent d'aquesta manera la construcció de xarxes de processat arbitràriament complexes en base a l'agregació, connexionat i encapsulació de multitud de components més senzills.

Gràcies a l'entorn proposat per *Jini / Apache River*, es pretén aconseguir que les agregacions de components construïdes mitjançant l'eina puguin ser publicades com a un servei. Això significa que la resta d'entitats pertanyents a la federació poden fer un ús remot dels esmentats serveis sense coneixement del seu emplaçament físic d'execució ni dels seus detalls d'implementació. Els serveis han de ser reutilitzables i han de permetre l'ús concurrent per part de diferents clients.

D'aquesta manera, les caixes negres mostrades per la interfície d'usuari de l'eina d'assemblatge poden representar processos software que s'executen a la màquina local o a qualsevol altre punt de la federació. És possible la interacció entre ambdós tipus d'agregacions de components i inclús la seva reutilització i publicació com a servei totalment nou. L'eina d'assemblatge és capaç de visualitzar i obtenir els serveis disponibles a la federació de la mateixa manera que visualitza els components software propis.

L'objectiu secundari d'aquest projecte és la millora constant de les funcionalitats ofertes per l'eina d'edició de components. A més a més de l'adaptació i optimització de la interfície gràfica per a facilitar l'ús de les possibilitats de publicació i obtenció de serveis, s'han implementat multitud de noves característiques que complementen el nou procés de construcció i desplegament de serveis i l'enriqueixen amb flexibles i potents possibilitats.

A l'acabament d'aquest projecte es disposa d'una eina versàtil capaç de construir xarxes de processat complexes com a resultat de l'agregació de multitud d'altres xarxes i serveis que operen en punts molt diversos de la federació d'entitats *Jini / Apache River*. És possible així mateix el desplegament i inicialització de les esmentades xarxes com a serveis públics en qualsevol punt de la federació habilitat a tal efecte, i el control i la gestió remota de tots els serveis publicats.

IV. ABSTRACT

In the present days, software tools that execute a distributed and decentralized process and data handling represent key figures for the good performance of a high number of industrial processes and business models. Specifically, Service Oriented Architectures (SOA) define an scenario where a set of entities belonging to a network collaborate among them with in an independent way to the underlying technology of each one of the entities, thus forming a flexible and auto-managed service federation.

The main goal of this project is the implementation of service oriented architecture in a software component editing and assembling tool provided with graphical interface. This goal is reached by means of the Jini / Apache River platform, a set of libraries and frameworks extending the native Java platform to ease and provide tools to deploy these kinds of architectures.

The starting point of the project is a simple software component assembling platform, entirely written in the Java programming language. This platform implements a user-friendly graphical interface to represent software components as black boxes that the user can interconnect to achieve their mutual collaboration; allowing in that way the creation and building of arbitrary complex processing networks as the aggregation, connection and encapsulation of many single components.

Thanks to the environment established by Jini / Apache River, this project aims to achieve the publication as services of the component aggregations built using that tool. The latter means that any of the entities belonging to the federation may invoke and use the services in a remote way, without knowledge about their physical place of execution or their implementation details. Services must be reusable and must allow concurrent utilization by a large number of users.

In that way, black boxes handled by the assembling tool's user interface may represent software processes running on the local machine or in any other place of the federation. Interaction between both types of component aggregations is possible, including their reutilization and publication as a totally brand new service. The assembling tool is capable of browsing and obtaining any service available within the federation in the same way that its own software components are shown and used.

The secondary goal of this project is the regular improvement of the functionalities offered by the component editing tool. As well as the adaptation and optimization of its graphical user interface to allow and ease the utilization of new service obtaining and publishing, many new features have been implemented to complement and improve the new process of service building and deployment, endowing this process with very flexible and powerful capabilities.

Upon the ending of this project a very versatile tool has been produced, with capability to build complex processing networks by means of the aggregation of many other networks and services that may operate in different and indeterminate points within the Jini / Apache River entities' federation. It is as well capable of performing the deployment and initialization of such networks as public and standalone services in any point within the federation endowed with the possibility of doing so. It also allows remote control and management of published services.

V. OBJETIVOS

Los objetivos principales de este proyecto se enumeran a continuación:

- Estudio y comprensión del entorno de programación *Jini / Apache River*, sus especificaciones y posibilidades.
- Estudio de la plataforma CAEAT: criterios de diseño software empleados, estructura, limitaciones, escalabilidad y posibilidades de ampliación.
- Ampliación del concepto de "bean", empleado por CAEAT, al concepto de "servicio", empleado en el entorno *Jini / Apache River*.
- Diseño de un esquema de publicación de servicios par las librerías de componentes de la plataforma CAEAT, suficientemente robusto y canónico como para ser fácil y rápidamente reproducible para todos los componentes a generar.
- Diseño e implementación de herramientas que permitan desplegar servicios *Jini / Apache River* desde la plataforma CAEAT, pudiendo formar diferentes instancias de CAEAT pertenecientes a una misma red una federación de servicios de manera sencilla y rápida.
- Implementación de un proceso de publicación que permita exportar como servicio cualquier agregación de componentes generada mediante la herramienta CAEAT.
- Diseño e implementación de un sistema eficiente de distribución de las librerías de componentes aptas para ser insertadas en la plataforma CAEAT y utilizadas para construir agregaciones. Introducción de un mecanismo para la protección de la libre distribución de dichas librerías.
- Implementación de los mecanismos necesarios para poder desplegar servicios de manera remota en cualquier punto de la federación *Jini / Apache River* habilitado para tal fin.
- Adaptación al nuevo entorno distribuido de los componentes ya existentes para la plataforma CAEAT. Creación de nuevos componentes, si se considera necesario.
- Implementación de los mecanismos necesarios para llevar a cabo un control y gestión de la vida de los servicios, pudiendo detenerlos, ampliarlos, editarlos, etc. a voluntad.
- Implementación de mecanismos de edición concurrente y en caliente de los servicios y agregaciones publicadas, sin necesidad de detener el procesado llevado a cabo por dichos servicios y garantizando la suficiente protección contra la modificación simultánea.
- Testeo de la aplicación finalizada en un entorno distribuido real, para garantizar su correcto funcionamiento en un escenario realista y para detectar los requisitos mínimos de funcionamiento en términos de configuración de red.

VI. GLOSARIO DE ACRÓNIMOS

AES: Advanced Encryption Standard

API: Application Programming Interface

ASF: Apache Software Foundation

CAEAT: Component and Aggregation Editing and Assembling Tool

CLE: CAEAT Library Encrypter

CNA: CAEAT Network Acceptor

CSM: CAEAT Service Manager

CVS: Concurrent Versions System

DES: Data Encryption Standard

DGC: Distributed Garbage Collector

DHCP: Dynamic Host Configuration Protocol

DNS: Domain Name System

FIFO: First In, First Out

GC: Garbage Collector

GUI: Graphical User Interface

IDE: Integrated Development Environment

IP: Internet Protocol

LAF: Look and Feel

JERI: Jini Extensible Remote Invocation

JRE: Java Runtime Evironment

JSP: JavaServer Pages

JVM: Java Virtual Machine

LIFO: Last In, First Out

LUS: Lookup Server / Lookup Service

MVC: Model – View – Controller

NAT: Network Address Translation

PAC: Programmable Automation Controller

PLC: Programmable Logic Controller

RMI: Remote Method Invocation

SCADA: Supervisory Control And Data Adquisition

SDK: Software Development Kit

SNC: SeNet Components

SOA: Service Oriented Architecture

SVN: Subversion

TCP: Transmission Control Protocol

UDP: User Datagram Protocol

VII. TABLA DE CONTENIDOS

VIII. ÍNDICE DE FIGURAS

IX. ÍNDICE DE EXTRACTOS DE CÓDIGO

1. ARQUITECTURA ORIENTADA A SERVICIOS, JINI Y APACHE RIVER

El presente proyecto tiene como principal objetivo la implementación de una arquitectura de proceso distribuida y orientada a servicios en una herramienta de ensamblaje y agregación de componentes software ya existente.

Este capítulo realiza una breve introducción a la filosofía de diseño, funcionamiento y despliegue de arquitecturas software orientadas a servicios. A continuación se describe la tecnología *Jini*, un conjunto de herramientas y especificaciones creadas sobre la plataforma *Java* que favorecen y facilitan la generación de arquitecturas como las descritas. Finalmente se hace mención de *Apache River*, la colección de librerías y recursos de distribución libre que actualmente implementan de manera práctica las directrices establecidas por *Jini*.

1.1. ARQUITECTURAS ORIENTADAS A SERVICIOS (SOA)

La arquitectura orientada a servicios ó *Service Oriented Architecture* (SOA) es una filosofía de diseño, implementación y despliegue software que surge a finales de los 90. Su principal motivación viene dada por las necesidades de los distintos modelos de negocio de disponer de sistemas de proceso distribuido simples, modulares e interoperables.

En un sistema desarrollado bajo la filosofía SOA, todas las actividades y procesos están diseñados para ofrecer un único (y simple) servicio, ya sea a otros servicios internos o a entidades externas (como por ejemplo, clientes). El diseño SOA enfatiza la interfaz entre los distintos servicios, no sus detalles de implementación, haciendo más ágiles y sencillas las tareas de actualización, modificación, etc. de dichos servicios.

La definición y objetivos de un "servicio" son independientes de la tecnología que se use para implementarlo. Un servicio es una operación sin estado y auto-contenida que acepta llamadas y devuelve unos resultados a través de una interfaz que debe ser bien conocida por la entidad que realiza la llamada. Un servicio se da a conocer públicamente mediante su interfaz, y admite re-utilización como cualquier otro recurso y utilización concurrente debido a su falta de estado y a su independencia respecto del estado de otros servicios. La llamada a un servicio contiene toda la información necesaria para que éste pueda realizar sus operaciones y devolver unos resultados.

Como se puede observar en la Figura 1.1, multitud de entidades distintas, trabajando bajo tecnologías muy diversas, pueden llegar a interactuar fácilmente en el marco de una arquitectura orientada a servicios dado que únicamente necesitan conocer la interfaz del servicio que pretenden utilizar para poder emplearlo, abstrayéndose de la naturaleza de las plataformas subyacentes.

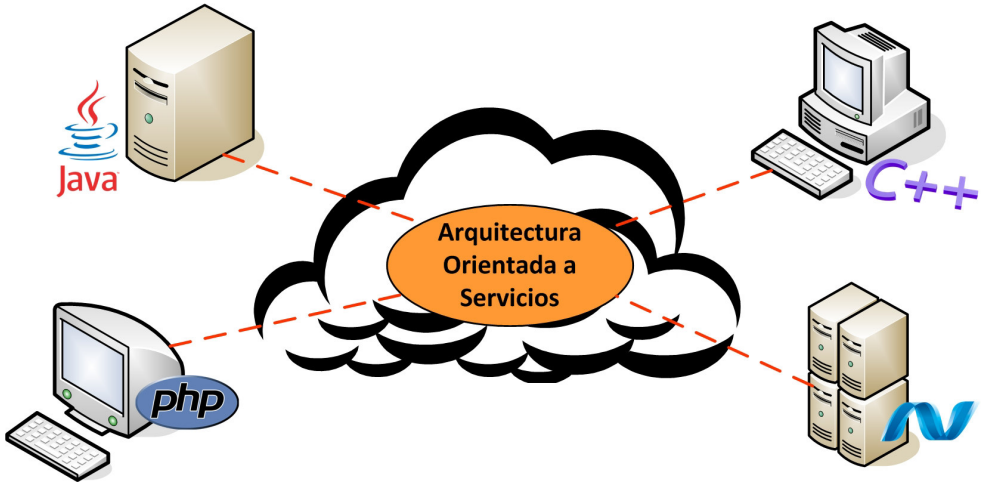

Figura 1.1: Representación de una Arquitectura Orientada a Servicios

La arquitectura orientada a servicios apunta hacia una estructura federativa del software, desde el momento en que promueve activamente la colaboración entre servicios y la auto-configuración y auto-gestión de los mismos. Un servicio pone a disposición de los potenciales clientes a través de sus interfaces una serie de componentes y recursos software. Estos recursos pueden ser usados por los clientes, o incluso por otros servicios, para desarrollar *ad hoc* aplicaciones que hagan uso de ellos.

Se establece de esta manera una clara separación entre dos entidades: las entidades proveedoras de servicios y las entidades consumidoras.

- **Proveedoras de servicios**: entidades que brindan el servicio a las entidades consumidoras, devolviendo a éstas el resultado adecuado al finalizar. Deben encargarse de darse a conocer adecuadamente a todas las potenciales consumidoras para que éstas puedan realizar las llamadas que estimen pertinentes.
- **Consumidoras de servicios**: entidades que realizan la llamada de petición de servicio a las proveedoras. Dicha llamada debe encapsular obligatoriamente toda la información necesaria para obtener los resultados deseados. Las entidades consumidoras deben disponer de un mecanismo para descubrir y conocer el conjunto de proveedoras a las que pueden tener acceso.

El paradigma de la arquitectura orientada a servicios es disponer de un tejido de proveedores y consumidores tal que la red sea lo suficientemente dinámica como para ser auto-gestionable y auto-suficiente, una característica conocida como *Spontaneous Networking*. Los servicios pueden ser desplegados ó finalizados de manera dinámica y sin afectación al resto de servicios, que se reconfigurarán adecuadamente. Los servicios deberán estar suficientemente redundados como para que la federación de servicios resultante sea fiable y pueda adaptarse de forma flexible a los cambios y requisitos de un entorno computacional dinámico.

En resumen, las ventajas que ofrece la adopción de un modelo de proceso basado en SOA son las siguientes:

- Mejora en los tiempos de actualización, modificación o reemplazo de procesos. Dada la independencia entre servicios, es posible actuar sobre uno de ellos sin que dicha actuación tenga afectación sobre los otros.
- Agilización y simplificación de todos los modelos de negocio que implican una interacción ó colaboración con otros entes a través de la red.
- Facilidad para desacoplar las capas de proceso y aplicación del sistema, pudiendo realizar actuaciones sobre la capa de aplicación sin tener para ello que interrumpir todo el modelo de negocio.
- Posibilidad de integración de tecnologías totalmente diferentes e incompatibles entre sí. Posibilidad de reciclaje de tecnologías obsoletas.
- Esquema de despliegue *plug & play*: es posible la agregación de nuevos servicios en caliente, ya que la característica de auto-gestión de la red provoca que estos sean visibles y aceptados por los servicios ya presentes en la red.
- Gran escalabilidad, ya que es posible federar un conjunto de servicios sencillos en uno más grande (y persistiendo en esta estrategia recursivamente, éste a su vez puede ser reutilizado por servicios más complejos).

1.2. JINI

Jini es la aproximación desde el lenguaje de programación *Java* a las arquitecturas orientadas a servicios descritas en el apartado anterior. Se trata de una API (*Application Programming Interface*) desarrollada por *Sun Microsystems* en 1998 que extiende la tecnología *Java* para facilitar la construcción de sistemas distribuidos consistentes en la federación de servicios dentro del marco de una red flexible y autogestionada.

El objetivo de *Jini* es transformar las redes en herramientas dinámicas en las que los recursos y los servicios puedan ser fácilmente encontrados y utilizados tanto por clientes humanos como por clientes computacionales. *Jini* permite la puesta en marcha y eliminación de servicios de una forma flexible y sencilla, permitiendo a todas las entidades involucradas unirse o abandonar la federación en el momento que lo consideren oportuno y de forma transparente al resto de entidades.

Asimismo, *Jini* persigue una estructura de red en la que dispositivos y componentes software de naturaleza muy diversa colaboran como si de un único sistema dinámico distribuido se tratase. Una federación *Jini* no debe ser vista como un conjunto rígido de clientes, servidores, usuarios y programas. En lugar de eso, debe ser imaginado como un conjunto de servicios (ya sean varios o únicamente uno) que pueden ser obtenidos para la realización de una tarea. La entidad cliente de esos servicios puede ser a su vez la proveedora de otros servicios que reutilicen a los primeros.

De todo lo anterior se desprende que la naturaleza flexible de *Jini* permite la construcción dinámica de servicios complejos y la desaparición de servicios que ya no se deban usar, requisito fundamental de una arquitectura orientada a servicios. Por tanto, la federación en sí misma es un ente que evoluciona y madura con el transcurrir del tiempo. Una federación *Jini* es vista desde fuera como un todo, por lo que se consigue hacer abstracción de la entidad física en la que residen los servicios, o el lugar en el que viven los recursos que se ponen a disposición de los clientes.

En los apartados siguientes se desglosan las diferentes entidades que forman parte de una federación de servicios *Jini*, así como los procesos que se llevan a cabo para iniciar, utilizar, detener, etc. un servicio. Se hace especial hincapié en cómo se concretan los conceptos abstractos expuestos en el apartado anterior en la solución particular adoptada por *Jini* y el lenguaje de programación *Java*.

Las especificaciones oficiales y completas de la plataforma *Jini* pueden ser consultadas en la referencia [**1**].

Figura 1.2: Logotipo de la plataforma *Jini* creado por Sun Microsystems

1.2.1. Definición de servicios Jini

El concepto más importante detrás de la arquitectura *Jini* es el de “servicio”. *Jini* define un servicio como una entidad que puede ser usada por una persona, un programa u otro servicio. Un servicio puede ser de naturaleza muy diversa: un componente software que realiza una tarea de computación, un espacio de almacenamiento, un canal de comunicación hacia otro usuario o servicio, una funcionalidad ofrecida por un componente hardware, etc.

Jini ofrece mecanismos y protocolos para que los servicios puedan "publicarse", es decir, darse a conocer en la red y ponerse a disposición de los potenciales clientes para su utilización. También ofrece a los proveedores de servicios herramientas para controlar el tiempo durante el cual éstos estarán a disposición de los clientes. Es posible declarar un servicio como caducado cuando se desee, provocando que éste deje de formar parte de la federación, o actualizar el servicio o añadir servicios nuevos cuando el proveedor lo desee. Desde el punto de vista de los clientes o consumidores de servicios, *Jini* ofrece la posibilidad de sondear la red en busca de servicios que se adecuen a las necesidades de cada momento, obtener sus interfaces y utilizarlos a través de ellas. Los siguientes apartados describen los detalles de implementación de los protocolos y mecanismos mencionados.

Dado que *Java* es un lenguaje de programación orientado a objetos, la implementación última de un servicio en el lenguaje de programación *Java* consiste en objetos *Java*. De esta manera, un servicio no es más que una agrupación de objetos, más o menos compleja, que implementa una interfaz del lenguaje de programación *Java*. La interfaz define completamente las operaciones que el consumidor del servicio puede demandar, la información que se debe proporcionar en dicha demanda y los resultados que se esperan obtener por parte del servicio.

En el siguiente extracto de código se puede observar una sencilla interfaz del lenguaje de programación *Java*. Esta interfaz obliga a la implementación de tres métodos. El primero de ellos devuelve un objeto bajo petición, el segundo acepta un objeto como parámetro, mientras que el tercero acepta dos valores enteros y devuelve un nuevo valor entero, presumiblemente tras realizar algún cálculo interno.

```
public interface Interfaz_Sencilla {

      Object getValor();
      void setValor(Object valor);
      int calcularValor(int a, int b);


}
```

Código 1.1: Ejemplo de interfaz del lenguaje de programación *Java*

Nótese que esta es la única información que necesita un cliente para hacer uso del servicio. Con la interfaz anterior en su poder, el consumidor del servicio es consciente de las operaciones que puede realizar, los valores que debe proporcionar al proveedor y los resultados que éste le devolverá. No es conocedor, y aquí radica uno de los puntos fuertes de la filosofía SOA, de los detalles de implementación de cada uno de los métodos anteriores. Puede ser perfectamente posible que alguno de los métodos delegue el trabajo recursivamente en otros servicios, que podrían llegar a ser incluso ajenos a la tecnología *Jini* y al lenguaje de programación *Java*.

En el apartado anterior se ha remarcado que en una SOA los consumidores de servicios únicamente necesitan conocer la interfaz de dichos servicios para poder hacer uso de ellos. La estrategia que sigue *Jini* para poder cumplir este requisito consiste en obligar a los servicios a implementar una interfaz como la anterior y hacerla pública, de manera que todos los potenciales clientes del servicio sean conscientes de las funcionalidades (o métodos del lenguaje de programación *Java*) de las que pueden hacer uso, sin necesidad de conocer los detalles de implementación de dichas funcionalidades o la entidad física en la que se están llevando a cabo.

1.2.2. Lookup Servers

Hasta el momento, se ha hablado de dos entidades clave en el despliegue de arquitecturas orientadas a servicios: los proveedores de servicios y sus consumidores. La tecnología *Jini* introduce una tercera figura que resulta de importancia capital para que proveedores y consumidores puedan conocerse e interactuar unos con otros: los servidores de *Lookup* (LUS).

Un servidor de *Lookup* no es más que otro servicio *Jini* que forma parte de la federación de servicios. Su particularidad radica en que su función es la de mapear interfaces que determinan las funcionalidades ofrecidas por los servicios con grupos de objetos *Java* que implementan dicho servicio. En otras palabras, los servidores de *Lookup* hacen las funciones de "catálogo de servicios" al cual proveedores y consumidores pueden acceder con finalidades muy distintas que se detallan a continuación.

- **Proveedores de servicios**: se ha expuesto en apartados anteriores que en un ambiente SOA los proveedores de servicios deben dar a conocer éstos a los potenciales clientes. La estrategia adoptada por *Jini* para tal fin pasa por el uso de servidores de *Lookup*. En el momento de iniciar un servicio, la entidad proveedora contacta con uno o más servidores de *Lookup* y registra en él o ellos una serie de objetos *Java* descriptores del servicio. Los detalles de este proceso se explican en los siguientes apartados.
- **Consumidores de servicios**: cuando una entidad consumidora desea hacer uso de un servicio en concreto, debe ser capaz de sondear la red en busca de los servidores de *Lookup* disponibles y de recibir información sobre los objetos *Java* registrados en cada uno de ellos. La entidad cliente escogerá el objeto seleccionado, que le será transmitido por parte del LUS. A través del objeto obtenido, la entidad consumidora puede invocar y hacer uso de las funcionalidades del servicio. Los detalles de implementación de este proceso se explican en los siguientes apartados.

Una federación de servicios *Jini* debe contar, en consecuencia, con un tejido de servidores de *Lookup* suficientemente vasto y accesible como para que permanentemente se cuente con la redundancia de servicios deseada y se garantice una alta disponibilidad del servicio. La gran flexibilidad ofrecida por *Jini* radica en que no importa qué entidades ejecuten los servidores LUS, siempre que éstos sean visibles al resto de miembros de la federación. Se incentiva la diversidad de roles, de manera que una misma entidad puede estar desempeñando los siguientes papeles simultáneamente:

- Proveedora de una serie de servicios inicializados en ella
- Huésped de una o más instancias de servidores de *Lookup*
- Cliente de uno o más servicios ofrecidos por terceras entidades

El hecho de que una misma entidad desempeñe los tres roles descritos de manera simultánea pasa desapercibido para el resto de entidades de la federación, que hacen uso de los servicios y los servidores de *Lookup* con total abstracción del lugar físico en el cual se están ejecutando cada uno de estos procesos.

No es necesario implementar un servidor de *Lookup* para poder desarrollar aplicaciones en entornos basados en *Jini*. La distribución de la plataforma *Jini*, llevada a cabo en la actualidad por la fundación *Apache*, incluye varias implementaciones de servidores de *Lookup* altamente configurables. El programador debe preocuparse de configurar los parámetros del LUS según sus necesidades, así como de que los servicios desarrollados se comuniquen correctamente con él siguiendo los protocolos y mecanismos descritos en los apartados subsiguientes.

1.2.3. TCP/IP, mensajes Multicast, JERI y serialización

Puede parecer que el escenario planteado hasta el momento presenta un cierto aire de utopía. Las arquitecturas SOA en general y *Jini* en particular prometen que un servicio puede ser lanzado y ejecutado en cualquier punto de la red y que los clientes pueden hacer uso de él haciendo total abstracción del lugar en el que residen.

Nótese que hasta el momento no se ha hablado de protocolos de red concretos. En el caso de TCP/IP, no se ha hecho referencia a direcciones IP o puertos. Las arquitecturas SOA pretenden que las entidades no "ataquen" directamente direcciones y puertos, sino que soliciten funcionalidades a la red y ésta responda.

Dado que la estructura de red actual está férreamente ligada a TCP/IP, una tecnología que plantee implantar una arquitectura orientada a servicios debe proponer estrategias para solventar sus limitaciones y conseguir abstraerse del protocolo de red usado. *Jini* consigue este objetivo mediante las siguientes tres técnicas:

- Uso de mensajes *multicast* a través de la red
- JERI (*Jini Extensible Remote Invocation*)
- Serialización de objetos Java para transmitirlos a través de una red

A continuación se detalla brevemente la forma en que *Jini* hace uso de los tres ítems anteriores.

1.2.3.1. Mensajes multicast

En un entorno IP, existen tres estrategias básicas a la hora de transmitir un paquete desde un emisor a uno o varios receptores.

- ***Unicast***: el emisor envía el paquete directamente al *host* receptor usando su dirección IP.
- ***Broadcast***: el emisor envía el paquete a la dirección de *broadcast* de la sub-red. Los destinatarios de un *broadcast* son todos los *hosts* de la sub-red. Los nodos intermedios se hacen cargo de retransmitir copias del paquete a todos los *hosts* de la sub-red.
- ***Multicast***: el emisor envía el paquete a una dirección reservada para funciones *multicast*. Los destinatarios de una transmisión *multicast* son un sub-grupo de *hosts* de la sub-red. El paquete transmitido encapsula información sobre el grupo *multicast* al cual va dirigido el paquete. Cada host de la sub-red pertenece a uno, varios o ningún grupo *multicast*, de manera que los nodos intermedios se encargan de retransmitir el paquete únicamente a los *hosts* interesados en él.

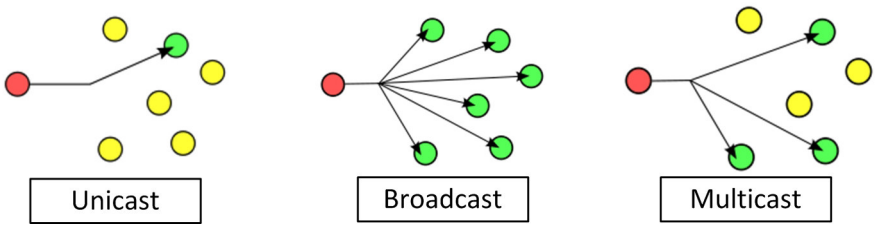

Figura 1.3: Estrategias de transmisión de paquetes en redes IP

Jini hace uso de mensajes *multicast* en los procesos de comunicación con los *Lookup Servers*. Cada LUS está asociado a uno o más grupos *multicast* configurables en el momento del despliegue, de esta manera puede realizarse una separación de los servidores atendiendo a los criterios que se deseen. El grupo *multicast* no es más que un nombre que se asigna a los *hosts* y que va encapsulado en los paquetes *multicast* que circulan por la red, resultando por lo tanto una propiedad fácilmente configurable y altamente intuitiva.

Cuando una entidad desea comunicarse con un grupo de *Lookup Servers*, ya sea para buscar servicios o para registrar uno nuevo en ellos, la única dirección IP que se necesita usar es la dirección *multicast* adoptada por las especificaciones de *Jini*, ya integrada en las API's de desarrollo. Cuando una entidad cliente obtiene un servicio, tampoco es necesario que conozca la dirección IP de la máquina en la que reside dicho servicio, como se detallará en el apartado siguiente.

1.2.3.2. JERI (Jini Extensible Remote Invocation) y RMI (Remote Method Invocation)

Desde el punto de vista del lenguaje de programación *Java*, el objetivo de *Jini* es poner a disposición de una JVM (*Java Virtual Machine*) recursos y objetos que le son ajenos y que están disponibles en la red. El modelo de programación de *Java* cuenta desde su versión 1.1 con las API's de RMI (*Remote Method Invocation*), que se encargan precisamente de esta tarea.

RMI facilita la movilidad de código entre diferentes Máquinas Virtuales de *Java*. Mediante RMI, es posible obtener una interfaz remota hacia una aplicación *Java* que se está ejecutando en una máquina virtual distinta. Esta interfaz no es más que un objeto que hace las funciones de *proxy*. Encapsula y esconde frente a los clientes los detalles de implementación (por ejemplo, la dirección IP y el puerto en el que la máquina remota está operando) y contiene la definición de los métodos a los que el cliente puede tener acceso a través de ella. De hecho, una interfaz remota RMI sería exactamente igual a la sencilla interfaz presentada en el extracto de código 1.1, con el añadido de que extendería a la interfaz `Remote`, contenida en las librerías de RMI, para indicar que las llamadas a las funciones que se definen en ella se hacen de manera remota.

```
public interface Interfaz_Sencilla extends Remote {

      Object getValor() throws RemoteException;
      void setValor(Object valor) throws RemoteException;
      int calcularValor(int a, int b) throws RemoteException;


}
```

Código 1.2: Ejemplo de interfaz remota RMI que los objetos *proxy* deberán implementar

Dado que el objeto *proxy* obtenido debe implementar necesariamente esta interfaz, el cliente que lo obtenga tiene completa información de los métodos a los cuales puede tener acceso, los parámetros que le debe proporcionar a cada uno y los resultados que le devolverán una vez finalizado el servicio. Cabe destacar que el concepto de "llamada remota" implica que todo el procesado se realiza en la máquina proveedora del servicio. El cliente se limita a llamar a la función a través del objeto *proxy* proporcionándole los parámetros y a esperar que el proveedor del servicio le devuelva el resultado, si lo hay.

En la figura siguiente se muestra un sencillo diagrama de secuencia que detalla los pasos que un cliente llevaría a cabo para invocar uno de los métodos remotos de la interfaz mostrada en el ejemplo anterior, proporcionándole sendos parámetros (los enteros 3 y 4) y recoger el resultado retornado (el entero 12). Nótese que no se hace hincapié en los mecanismos usados por RMI para obtener la interfaz remota, ya que *Jini* utiliza su propia metodología basada en la utilización de servidores de *Lookup* que se ha explicado en el apartado anterior.

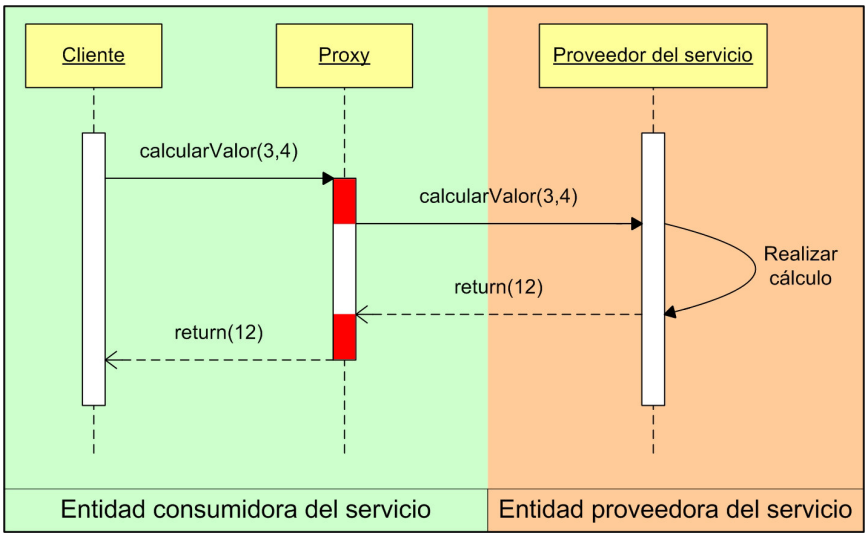

Figura 1.4: Ejemplo de llamada remota a una función a través de un objeto *proxy*

Se debe destacar también que el objeto *proxy* que implementa la interfaz remota no debe ser visto como un mero "puente" entre cliente y servidor. Dicho objeto puede ejecutar código previamente a realizar la llamada remota sobre el servidor y antes de retornar al cliente los resultados (en el diagrama, en los intervalos representados en rojo). Esta técnica ofrece una potencia y una libertad adicional en esquemas distribuidos como los descritos, pues los datos que intercambian clientes y servidores pueden ser adaptados y procesados como se desee sin que ni unos ni otros deban preocuparse de hacerlo. Cuando un objeto *proxy* ofrece capacidad para realizar computación se define como *smart proxy*. Esta característica es de suma importancia en el presente proyecto.

Como se puede deducir de esta explicación, el entorno que ofrece RMI se adapta a la perfección a una estructura de red como la propuesta por *Jini*. Sin embargo, la tecnología *Jini* en su versión 2.0 no utiliza únicamente la semántica que ya definió RMI en su momento para construir su federación de servicios, sino que la mejora. En la actualidad *Jini* define JERI (*Jini Extensible Remote Invocation*), una librería de clases y utilidades construidas sobre RMI que la mejora ostensiblemente en los siguientes aspectos:

- Soporte sencillo y transparente para más protocolos de transporte al margen de TCP. RMI ha generado con el paso del tiempo nuevas implementaciones sobre otros protocolos, pero implicaban cambiar por completo la interfaz de programación, introduciendo nuevas clases y librerías cada vez.
- Más posibilidades de customización de los parámetros usados a la hora de exportar un servicio y generar su objeto *proxy*.
- Un sistema de *Distributed Garbage Collection* más flexible y eficiente. El *Garbage Collector* es el proceso residente en toda Máquina Virtual de *Java* que se encarga de liberar la memoria de los objetos que han llegado al final de su ciclo de vida y no serán usados por ningún otro objeto. DGC es su versión para sistemas distribuidos.
- Mejora en la gestión de la seguridad. Se ponen a disposición nuevos mecanismos de autenticación y verificación de la integridad de los objetos descargados de manera dinámica.

En la siguiente figura se muestran las diferentes capas del entorno *Jini* y la plataforma *Java* implicadas en la utilización remota de un servicio. Como se puede ver, JERI es un protocolo que hace uso del RMI nativo de la plataforma *Java* para adaptarlo a la arquitectura propuesta por *Jini* y dotarlo de todas las nuevas y potentes funcionalidades que se acaban de exponer. El nivel más bajo de la torre de protocolos lo marca la capa de transporte. Como se ha explicado, la utilización de un servicio *Jini* se inicia con el envío de mensajes *multicast* a este nivel.

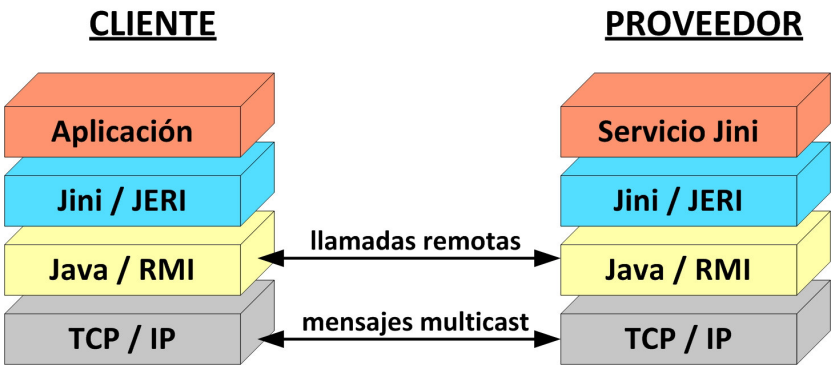

Figura 1.5: Diagrama de capas y protocolos implicados en la utilización remota de un servicio *Jini*

1.2.3.3. Serialización de objetos Java

Los objetos creados por la Máquina Virtual de *Java* durante la ejecución de un programa existen en memoria únicamente durante el tiempo de vida de dicha máquina virtual. La serialización es un procedimiento, no exclusivo de Java, por el cual un objeto puede ser transformado en una ristra de *bytes* que se puede restaurar dando lugar al objeto original a posteriori.

La serialización puede ser útil, por ejemplo, para salvar las estructuras de datos de un programa a un fichero y restaurarlas en una ejecución posterior. También sirve para transmitir objetos "vivos" a través de la red: la Máquina Virtual de *Java* receptora es capaz de reconstruir el objeto a partir de la tira de *bytes* en la que ha sido convertido. Esta última funcionalidad es usada por *Jini* para conseguir la movilidad de objetos por la red que se ha explicado hasta el momento.

Existen, no obstante, dos requisitos que se deben cumplir para que un objeto pueda ser apto para ser serializado y transmitido por la red. El primero de ellos consiste en la implementación de la interfaz `Serializable` de *Java*. Los objetos creados a partir de clases de *Java* que implementan dicha interfaz son serializables, siempre que dentro de ellos no encapsulen otros objetos que no lo sean. La gran mayoría de las clases básicas del lenguaje de programación Java son aptas para serialización.

El segundo requisito guarda relación con la manera en que los objetos son serializados y transmitidos. Únicamente se transmite el estado del objeto, el valor de sus propiedades en el momento de la serialización. Se da por supuesto que la máquina virtual receptora conoce la clase de la cual el objeto es una instancia. En el lenguaje de programación *Java*, la JVM únicamente puede utilizar objetos que sean instancias de clases cuya definición haya sido cargada desde el entorno local (es decir, clases que sean "bien conocidas" por la JVM que ejecuta el programa).

En un entorno distribuido la condición anterior no se suele cumplir: la JVM receptora posiblemente ignore por completo la naturaleza de las clases que han generado los objetos recibidos. Es por ello que la JVM que transmite los objetos debe poner de algún modo al alcance de la JVM receptora el código fuente que ha generado los objetos transmitidos. *Jini* utiliza las mismas soluciones que adoptó RMI para tal fin. Estos mecanismos son detallados con más profundidad en el capítulo 3.3.2, en el cual se expone el esquema aplicado para solventar estas situaciones en el presente proyecto.

Las clases cuyos objetos son susceptibles de ser serializadas deben declarar un atributo numérico llamado `serialVersionUID` que sirve a la JVM para llevar un control de las versiones de los objetos serializados. Esto sirve para garantizar que la clase que ha generado el objeto serializado y la clase conocida por la JVM que está de-serializando y utilizando el objeto son idénticas.

Las especificaciones completas de las API's del lenguaje de programación *Java* que se encargan de la serialización de los objetos se pueden consultar en la referencia [**2**].

1.2.4. Proceso de publicación y obtención de un servicio

En este apartado se describen brevemente los procedimientos y protocolos que la plataforma *Jini* utiliza para que los proveedores de servicios puedan registrar los objetos necesarios en los servidores LUS; así como los mecanismos que llevan a cabo los clientes para buscar y obtener estos objetos y hacer uso del servicio a través de ellos.

La visión que se ofrece de dichos mecanismos y protocolos es la mínima imprescindible para comprender los conceptos usados durante la elaboración del presente proyecto. Para obtener una exposición más exhaustiva de los mismos, consúltese la documentación oficial de la plataforma Jini en la referencia [**1**].

1. **Situación inicial**: por simplificación, en este apartado se va a considerar una red formada únicamente por los 3 miembros principales de una federación *Jini*: el proveedor del servicio, el servidor de *Lookup* y el cliente. Se debe recordar que esta no es la situación normal, ya que *Jini* promueve precisamente la variedad de roles en la red. Es posible que la entidad que publica el servicio en el LUS no sea la misma que lo ofrece. También es posible que la entidad que está ejecutando el LUS sea la misma que la que provee el servicio. Y por descontado, en un entorno real casi con toda probabilidad el servidor de *Lookup* se encontraría redundado.

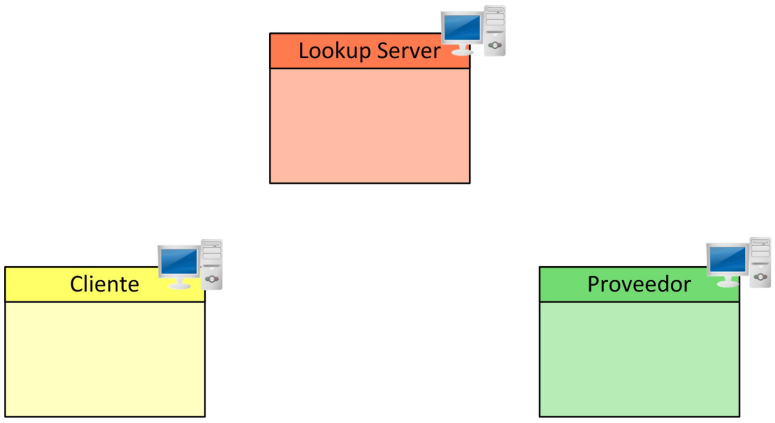

Figura 1.6: Escenario de partida para los ejemplos siguientes

2. **Exportación del servicio**: la máquina proveedora del servicio lo exporta para su uso remoto. El concepto de "exportación" implica tomar un objeto o conjunto de objetos *Java* y asociarlos a un puerto de la máquina local sobre el cual se restará a la escucha de peticiones de servicio por parte de los clientes. Estas peticiones consistirán en llamadas remotas a las funciones definidas en la interfaz del servicio, que llegarán a través del objeto *proxy* obtenido por los clientes. La máquina proveedora del servicio atenderá peticiones remotas de servicio hasta que el objeto ó grupo de objetos sean convenientemente des-exportados y por lo tanto el servicio se dé por finalizado.

3. **Creación del *proxy***: a partir de la agrupación de objetos exportada, se creará un objeto *proxy* que se registrará en los servidores de *Lookup*. Como se ha expuesto en apartados anteriores, este objeto deberá implementar los métodos que se deseen poner a disposición de los clientes. El objeto *proxy* encapsula los detalles de implementación de la comunicación con el servidor; por ejemplo, la dirección IP y el puerto en el cual la máquina proveedora se encuentra ofreciendo el servicio. Cabe destacar el hecho de que el objeto *proxy* no sólo encapsula estos datos, sino que los "esconde" de cara al cliente, que nunca llega a tener conocimiento explícito de éstos al ser el *proxy* la entidad encargada de la comunicación con el servicio real. Éste es un requisito imprescindible en un entorno que siga la filosofía de SOA.

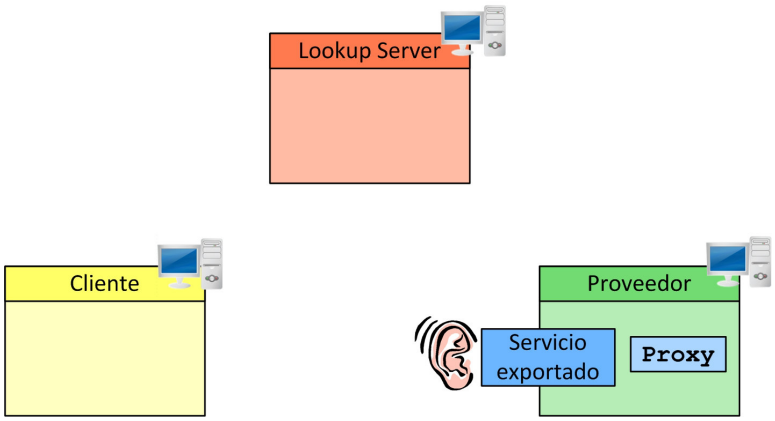

Figura 1.7: Exportación del servicio y creación de su *proxy* por parte de la máquina proveedora

4. ***Discovery***: a continuación la máquina proveedora del servicio realizará un *discovery* para tener conocimiento de los servidores LUS presentes en la red. El proceso de *discovery* consiste en el envío redundado de mensajes *multicast* dirigidos al grupo *multicast* del cual se desea recibir respuesta. La siguiente figura muestra el estado del escenario de prueba elegido una vez llevados a cabo los pasos anteriores.

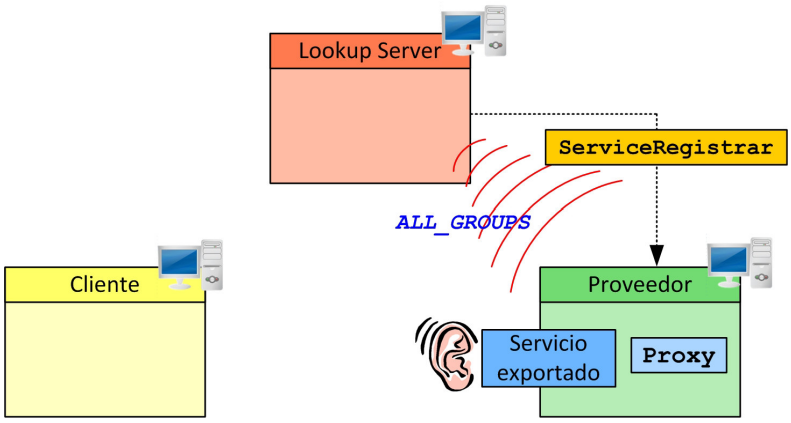

Figura 1.8: Protocolo de *discovery* y obtención de los *ServiceRegistrar*

En el presente ejemplo, el proceso de *discovery* se ha llevado a cabo sobre el super-grupo "ALL_GROUPS", usado para incluir a todos los grupos *multicast* presentes en la red. Al recibir la petición y decidir que es uno de los servidores aludidos, el servidor de *Lookup* observa la dirección IP encapsulada en el mensaje de *discovery* y devuelve a dicha máquina un objeto `ServiceRegistrar` que encapsula los detalles de comunicación con el LUS. El objeto `ServiceRegistrar` puede verse como el *proxy* del servidor LUS que ayuda a hacer abstracción de sus detalles de implementación: dirección IP, puerto, etc.

5. ***Join* (registro):** el objeto `ServiceRegistrar` actúa de *proxy* entre el LUS y las entidades que desean operar con él, ofreciendo multitud de funcionalidades que se explican con todo detalle en la referencia [**1**]. Una de ellas consiste en la implementación del protocolo *Join*, mediante el cual es posible registrar el objeto *proxy* del servicio en el servidor LUS para ponerlo a disposición de los clientes. De manera no obligatoria, pero sí muy recomendable, es posible asociar a dicho *proxy* una serie de atributos (llamados atributos de *Entry*) que servirán para facilitar la clasificación y localización del servicio en un formato "agradable para los humanos". Estos atributos pueden ser desde datos básicos sobre el servicio (nombre, tipo, creador, versión, utilidad, etc.) hasta un icono descriptivo del mismo, pasando por números de serie identificativos del servicio.

El registro de objetos en los LUS no se realiza de manera permanente. El servidor LUS mantendrá el registro del objeto únicamente durante un periodo de tiempo determinado, denominado periodo de concesión o *lease*. Este tiempo depende de la implementación y la configuración concreta del LUS, así como del número de objetos que el LUS ya contenga registrados. Cuando el tiempo de concesión expire, el objeto *proxy* registrado será eliminado de la tabla de objetos del LUS.

Para dar la oportunidad al proveedor del servicio de extender esta concesión, el LUS devuelve a la entidad publicadora del servicio un objeto `Lease` que encapsula les detalles de esta concesión, como el tiempo restante de la misma ó el LUS que la ha concedido. Este objeto puede ser usado por la misma entidad que ha publicado el servicio ó ser transferido a otra distinta. En cualquier caso, el objeto `Lease` sirve para mantener el servicio activo y publicado en el servidor LUS. A través de él puede solicitarse periódicamente la renovación de la concesión cuando se detecta que ésta está próxima a su finalización. De la misma manera puede cancelarse la concesión a voluntad (y por lo tanto eliminar el objeto registrado del LUS permanentemente).

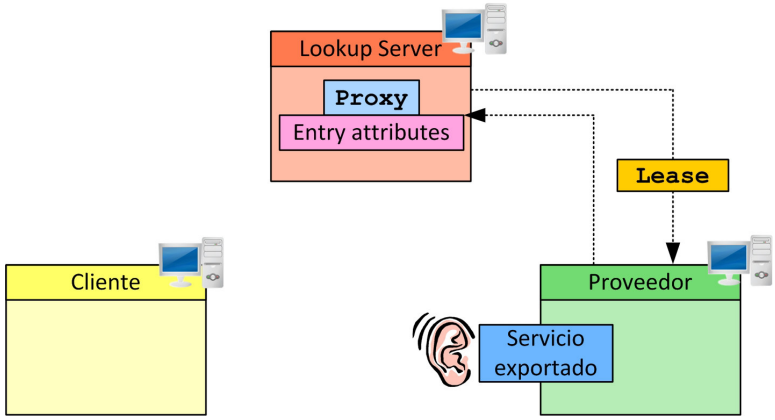

Figura 1.9: Registro de un objeto *proxy* y sus atributos en un servidor de *Lookup*

Cabe destacar que el servidor de *Lookup* asigna a cada objeto registrado en él un identificador universal llamado `ServiceID` que ayuda a diferenciar unívocamente cada servicio distinto dentro de una federación *Jini*. Para más información sobre la generación de dicho identificador, consultar el capítulo 3.3.2.

6. ***Lookup*:** tras realizar el registro, el servicio ya está a disposición de los clientes en el LUS durante todo el tiempo que dure la concesión. El próximo paso consiste en la búsqueda del servicio por parte de un cliente interesado en él. Nuevamente, para contactar con los servidores LUS se usarán paquetes *multicast* dirigidos al grupo deseado.

   Para localizar un servicio que se adecue a sus necesidades, el cliente puede usar plantillas mediante objetos de tipo `ServiceTemplate`. Estas plantillas pueden definirse de dos maneras distintas:

   - Si el cliente conoce la interfaz del servicio que desea utilizar, solicitará servicios que implementen dicha interfaz.
   - Si el cliente no conoce la interfaz y el servicio ha sido registrado con atributos de *Entry* (práctica muy recomendable), la plantilla puede incluir algunos de los atributos deseados. Este último método permitiría a las aplicaciones buscar servicios de una forma agradable para los humanos (por ejemplo, permitiendo al usuario introducir una palabra a partir de la cual se realizará la búsqueda).

   Independientemente de la técnica usada, la respuesta por parte del servidor LUS (si hay coincidencia entre la plantilla y los servicios registrados en él) es la devolución de un objeto

de tipo `ServiceItem`, que encapsula una copia del *proxy* del servicio además de otros parámetros útiles como el identificador universal `ServiceID`.

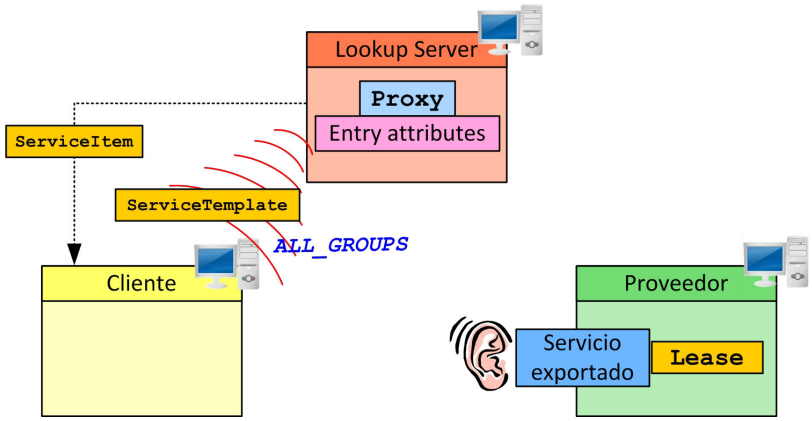

Figura 1.10: Proceso de *Lookup* iniciado por un cliente

7. **Utilización del servicio:** cuando el cliente des-encapsula el *proxy* y le da el tratamiento adecuado (algo que dependerá de la aplicación concreta que se esté usando), la figura del servidor de *Lookup* desaparece de la arquitectura. La comunicación se realiza entre cliente y servidor a través del *proxy* remoto, como se ha explicado en el apartado 1.2.3.2 dedicado a JERI, RMI y las llamadas a funciones remotas.

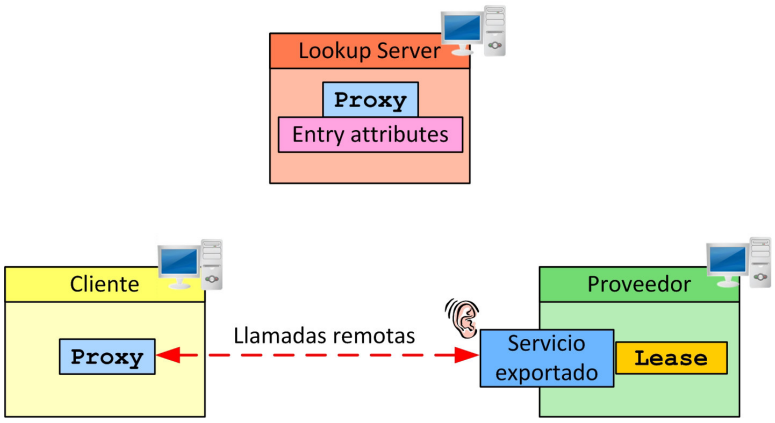

Figura 1.11: Comunicación cliente-servidor mediante invocaciones remotas

Por supuesto, la obtención del *proxy* del servicio por parte de un cliente no implica su consumo: el *proxy* continúa registrado en el servidor LUS y el cliente no recibe más que una copia de éste. Del mismo modo, el hecho de que un proveedor de servicios se encuentre atendiendo las llamadas remotas que le llegan por parte de un servicio no implica de ninguna manera que no pueda atender a otros clientes de manera concurrente. La concurrencia en el uso de los servicios es un requisito en un entorno SOA, y la problemática que presenta en el presente proyecto será ampliamente discutida en capítulos siguientes.

Nótese que el cliente realizará las llamadas sobre el *proxy* que habita en su entorno local como si todo el proceso se realizase en dicho entorno local. El cliente en ningún momento ha necesitado conocer la ubicación, direcciones IP, puertos, etc. de los servidores de *Lookup* ni del proveedor del servicio. Aunque el ejemplo se ha ilustrado con un servidor de *Lookup* y un proveedor de servicios, el cliente también hace abstracción del número de servidores y proveedores presentes en la red en el caso más común de que haya varios de ellos en ejecución. Se cumplen así los requisitos que se presentaron de manera teórica al exponer la filosofía de las Arquitecturas Orientadas a Servicios.

1.2.5. Controversia

La tecnología *Jini* ha sufrido diferentes críticas desde su creación. Las más recurrentes son las siguientes:

- **Necesidad de *Lookup Servers***: esta crítica está sustentada en el hecho de que *Jini* promete un entorno totalmente descentralizado tal y como definen las arquitecturas SOA. Sin embargo, la publicación y obtención de un servicio debe pasar necesariamente por, como mínimo, un *Lookup Server*. Este hecho es visto como un requisito que rompe la descentralización, pese a las enormes posibilidades de redundancia que ofrecen los servidores de *Lookup*.
- **Alcance de los paquetes multicast**: en una red IP, el alcance de los mensajes *multicast* usados por *Jini* no es ilimitado. Por lo general, el alcance de un paquete de tipo *multicast* o *broadcast* en una red IP está limitado a la misma sub-red desde la que se genera el paquete. Si se quiere un alcance mayor, los *routers* y *gateways* en los extremos de la red deben configurarse adecuadamente en el momento del despliegue para redireccionar este tipo de mensajes adecuadamente.
- **Necesidad de una JVM**: *Jini* es una tecnología desarrollada completamente en el lenguaje de programación *Java*, por lo que toda entidad que desee participar de una federación de servicios *Jini* deberá, como requisito imprescindible, disponer de una Máquina Virtual de *Java* instalada; independientemente de que a posteriori la tecnología permita interactuar con servicios basados en otras plataformas de desarrollo.

1.3. APACHE RIVER

En enero de 2010, *Sun Microsystems* fue absorbida por la compañía *Oracle*, adquiriendo ésta última los derechos sobre la mayoría de tecnologías desarrolladas por *Sun*. La plataforma *Jini* se distribuye en estos momentos bajo licencia *Apache* y se ha integrado dentro de un proyecto que se ha denominado *Apache River*. En el momento de la redacción del presente documento, la versión de *Apache River* bajo distribución es la 2.2.0.

Apache River incluye el conjunto de librerías llamado *Jini Starter Kit*, que contienen las clases originales de *Jini* necesarias para poner en marcha servicios dentro de una federación *Jini* así como para implementar aplicaciones cliente que hagan uso de ellos. Sin embargo, dado que el objetivo de *Apache River* es favorecer el despliegue sencillo de entornos distribuidos, el *framework* se distribuye junto con otras herramientas y utilidades que ayudan a conseguir tal fin. En el apartado 1.3.2 se detallan cuáles de estos complementos han sido de aplicación en la elaboración del presente proyecto.

En el sitio web de *Apache River* (ver referencia [**3**]) es posible obtener la versión actual de la plataforma, así como consultar las especificaciones de las distintas API's que la forman y la documentación oficial que soporta el proyecto.

Figura 1.12: Logotipo adoptado por *Apache* para el proyecto *River* tras la absorción de *Sun*

1.3.1. Licencia de distribución *Apache*

La licencia de distribución *Apache* (*Apache License*) es una licencia de software libre creada por la *Apache Software Foundation* (ASF). Como cualquier otro tipo de licencia de software libre, la *Apache License* permite al usuario del software la libertad de usarlo para cualquier propósito, redistribuirlo, modificarlo, e incluso redistribuir versiones modificadas de dicho software.

Multitud de tecnologías y plataformas de uso extendido en el momento de la elaboración del presente proyecto se distribuyen bajo la *Apache License*. Algunos ejemplos son:

- *Apache HTTP Server*, conocido servidor HTTP de código abierto
- *Android*, sistema operativo para dispositivos móviles basado en *Linux*
- *Spring*, un *framework* de código abierto de desarrollo de aplicaciones para la plataforma *Java*

Esta licencia no obliga a que las obras que hagan uso de software licenciado bajo la *Apache License* tengan que distribuirse bajo la misma licencia. Al contrario, no exige ni siquiera que dichas obras sean proyectos de código abierto, diferenciándose así del software distribuido bajo licencia *copyleft*. Este es el escenario en el que se encuentra el presente proyecto, en el cual se pretende en el futuro distribuir una plataforma de código cerrado que hace uso de software licenciado bajo la *Apache License* (las librerías de *Apache River*).

Sin embargo, esta licencia sí exige que en la distribución de software que haga uso de código licenciado bajo la *Apache License* se mantenga una noticia ó *disclaimer* que informe a los usuarios receptores que en el proyecto se ha usado código licenciado bajo la *Apache License*. El formato de este *disclaimer* viene definido en los términos y condiciones de la *Apache License*.

Por último, se debe destacar que la *Apache Software Foundation* no exige la distribución del código fuente en caso de modificación de software creado por la propia fundación, pero sí recomienda encarecidamente a los desarrolladores que remitan una copia a la ASF.

La *Apache License* ha conocido tres versiones distintas, la 1.0, la 1.1 y la 2.0. Ésta última es la versión actual en el momento de redacción del presente documento; fue redactada y aprobada en enero de 2004 y se pueden consultar sus términos y condiciones en la referencia [**4**].

1.3.2. Contenido de *Apache River*

Como ya se ha comentado anteriormente, la distribución completa de *Apache River* no incluye únicamente las API's que forman el *Jini Starter Kit*, sino que se adjuntan multitud de complementos y aplicaciones para facilitar el despliegue de arquitecturas distribuidas como las expuestas en este capítulo. A continuación se detallan cuáles de estas utilidades han sido usadas en la elaboración del presente proyecto:

- **Servidores de *Lookup***: la figura del servidor LUS es indispensable en una federación de servicios *Jini*. *Apache River* ofrece diferentes implementaciones de distintos tipos de servidores de *Lookup*. La implementación más sencilla, y también la que mejor se adapta a un entorno como el propuesto en el presente proyecto, recibe como nombre *Reggie*.
- **Servidor HTTP**: en el capítulo 1.2.3.3 se mencionó que para conseguir llamadas a métodos remotos a través de objetos *proxy* era necesario que la máquina proveedora del servicio pusiese a disposición del cliente el código fuente de las clases necesarias (este mecanismo se explicará con más detalle en capítulo posteriores). Para facilitar esto, *Apache River* ofrece un sencillo servidor HTTP elaborado en el lenguaje de programación *Java* fácilmente configurable para adaptarse a cualquier proyecto.

- **Archivos de configuración**: se adjuntan los archivos de configuración que usan las aplicaciones anteriormente descritas. Estos archivos incluyen todos los posibles parámetros configurables establecidos a un valor por defecto, permitiendo que el desarrollador, en el momento del despliegue, los sustituya por aquéllos que mejor se adapten a su entorno y objetivos.
- ***Scripts* ejecutables**: también se incluyen *scripts* ejecutables, tanto en versión *Windows* como *Unix*, que ejecutan automáticamente las aplicaciones anteriormente descritas usando los archivos de configuración adecuados. El objetivo es que el despliegue de una arquitectura basada en *Jini* / *Apache River* sea lo más rápida y limpia posible, y que únicamente sea necesario establecer la configuración de los diferentes "actores" la primera vez que éstos se ejecuten.
- **Ejemplos de utilización**: por último, la distribución también incluye algunos ejemplos fácilmente compilables y ejecutables para iniciar al desarrollador en el entorno *Jini*.

2. COMPONENT AND AGGREGATION EDITING AND ASSEMBLING TOOL

El presente proyecto tiene como principal objetivo la plasmación de una arquitectura orientada a servicios, tal y como la que define *Jini* / *Apache River* y se expone en el capítulo anterior, en una herramienta de ensamblaje y agregación de componentes software ya existente.

Este capítulo se dedica a la descripción de la herramienta CAEAT (*Component and Aggregation Editing and Assembling Tool*) en el estado de desarrollo en el cual se encontraba al inicio de la elaboración del presente proyecto. Se detallan sus utilidades y sus diferentes funciones y capacidades, a una profundidad suficiente para poder comprender las mejoras que ha supuesto la consecución de los objetivos que persigue el presente proyecto y que se desglosan en capítulos posteriores. También se explica la utilidad futura de una herramienta como CAEAT y las razones por las cuales encaja a la perfección en un entorno orientado a servicios como el que propone *Apache River* y se ha descrito ampliamente en el capítulo anterior.

Nota: el Capítulo 2 de esta memoria incluye capturas de la interfaz gráfica de la plataforma CAEAT tal y como ésta se encontraba al inicio de la elaboración del presente proyecto. Ésta es la manera más natural de exponer ordenadamente las características y funcionalidades del software, en un orden similar al que fueron cronológicamente creadas. La inclusión prematura de imágenes de CAEAT en su estado tras la finalización de este proyecto obligaría a introducir en este capítulo conceptos que se exponen más detalladamente en capítulos propios, haciendo farragosa en exceso la explicación breve de la naturaleza de la plataforma, que constituye el objetivo primordial de este capítulo.

2.1. DESCRIPCIÓN GENERAL DE LA HERRAMIENTA

La herramienta CAEAT forma parte de un proyecto más extenso denominado *SeNetComponents* (SNC). *SeNetComponents* es un proyecto que comprende la creación, edición, despliegue y ejecución de aplicaciones construidas en base a la agregación o unión de componentes software más sencillos. CAEAT, de las siglas en inglés *Component and Aggregation Editing and Assembling Tool* es, como su nombre indica, la herramienta que se ha diseñado e implementado para permitir la creación y edición de dichas agregaciones de componentes.

CAEAT es una herramienta desarrollada enteramente en el lenguaje de programación *Java* que permite la construcción de las susodichas agregaciones de componentes de una forma rápida, mediante una interfaz gráfica altamente intuitiva y sin necesidad de conocimientos de programación o de tener que generar código fuente durante el proceso.

Resulta necesario, como paso previo a la presentación de la herramienta, definir los conceptos de “componente” y “agregación de componentes”.

- **Componente**: se entiende por “componente” una pieza de software muy sencilla que realiza una función muy concreta. Un componente es reutilizable (filosofía del “ladrillo básico” con el cual se construyen distintos tipos de “edificios”) e invocable de manera concurrente, y se puede federar con otros componentes para formar entes más complejos que realicen funciones más avanzadas.
- **Agregación de componentes**: federación de componentes software, que pueden ser de naturaleza muy diversa, que forman una red que puede ser concebida como un super-componente que se sirve de los componentes sencillos para realizar funciones más complejas que éstos. Los componentes contenidos en ella están conectados y colaboran entre sí, en el sentido de que intercambian datos y hacen uso los unos de los otros, para conseguir realizar las funcionalidades para las cuales la agregación fue diseñada.

El principal objetivo y mérito de la plataforma CAEAT consiste en conseguir el manejo y ensamblaje de los ya mencionados componentes software de manera gráfica. CAEAT ofrece una interfaz

mediante la cual estos componentes software son representados sobre un tapiz como una "cajita negra con patillas". La caja negra, que el usuario puede manejar a voluntad, encapsula y esconde los detalles del componente software que está trabajando por debajo. Las patillas que salen de la caja representan la interfaz con el usuario y con el resto de componentes: expresan las propiedades y atributos del componente que se pueden manejar, y las acciones que se pueden llevar a cabo con él.

Figura 2.1: Representación gráfica en el tapiz de CAEAT de un componente

Un grupo de componentes en el tapiz de CAEAT pueden ser conectados entre sí mediante sus patillas. Este hecho se representa gráficamente como un "hilo" ó conexión entre las patillas de las dos cajas. A partir del momento de la conexión, ambos componentes software están compartiendo la propiedad ó característica representada por las patillas, y un cambio en dicha propiedad será transmitido a través de la conexión al resto de componentes conectados a esa propiedad. Los detalles de implementación de esta característica se detallan en el apartado 2.3, que versa sobre el estándar *JavaBeans*.

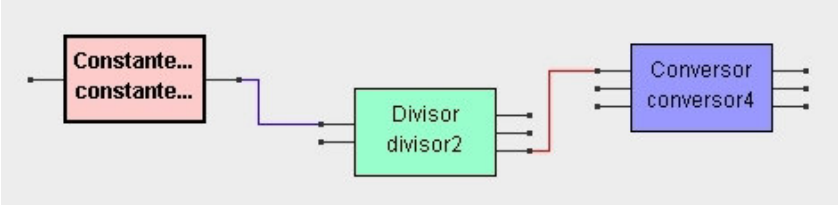

Figura 2.2: Interconexión de tres componentes en el tapiz de CAEAT

CAEAT también permite anidar diversos componentes dentro de un super-componente que los englobe. De esta forma, mediante la inserción de componentes en el tapiz, su interconexión y su anidamiento, es posible la construcción de agregaciones de componentes como las anteriormente descritas, que llevarán a cabo funciones arbitrariamente complejas a partir de la suma de todas las funciones sencillas desempeñadas por los componentes simples. Estas agregaciones también serán referidas en el presente documento como "redes" ó "esquemas", a causa de su aspecto visual en la superficie del tapiz de CAEAT.

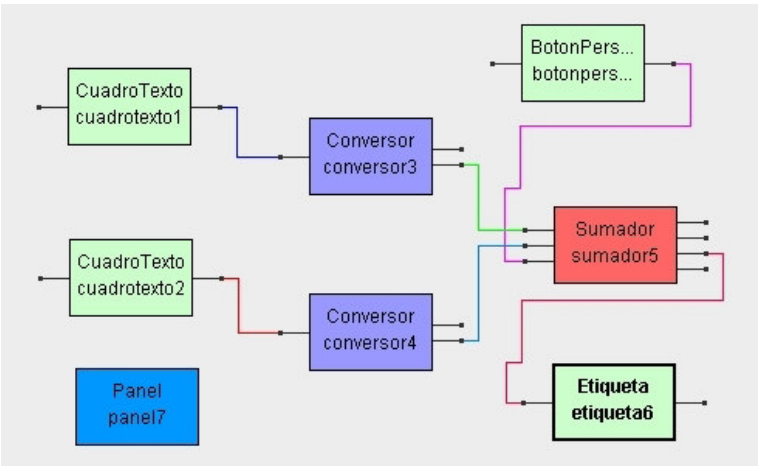

Figura 2.3: Ejemplo de agregación o esquema de CAEAT

2.2. INTERFAZ GRÁFICA Y FUNCIONALIDADES BÁSICAS

A continuación se muestra la interfaz gráfica de la plataforma CAEAT durante una sesión normal de trabajo:

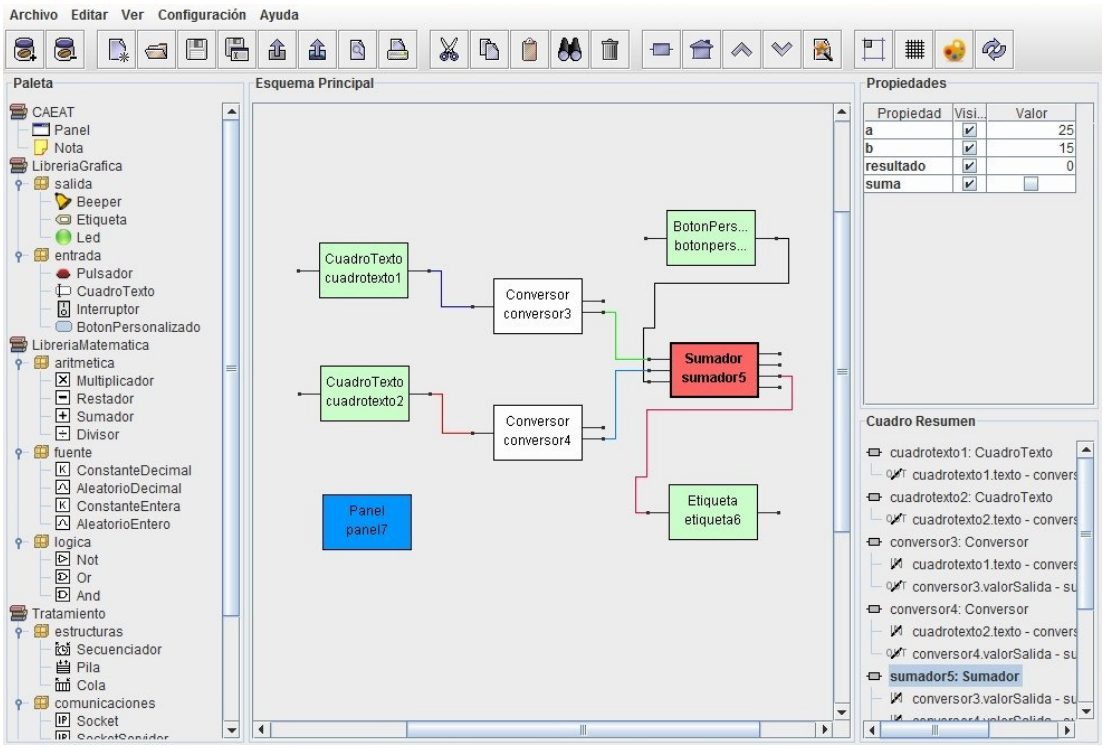

Figura 2.4: Interfaz de trabajo de la plataforma CAEAT

En los sub-apartados siguientes se describe muy brevemente la función de cada uno de los elementos de esta interfaz gráfica, que se puede dividir claramente como se muestra a continuación:

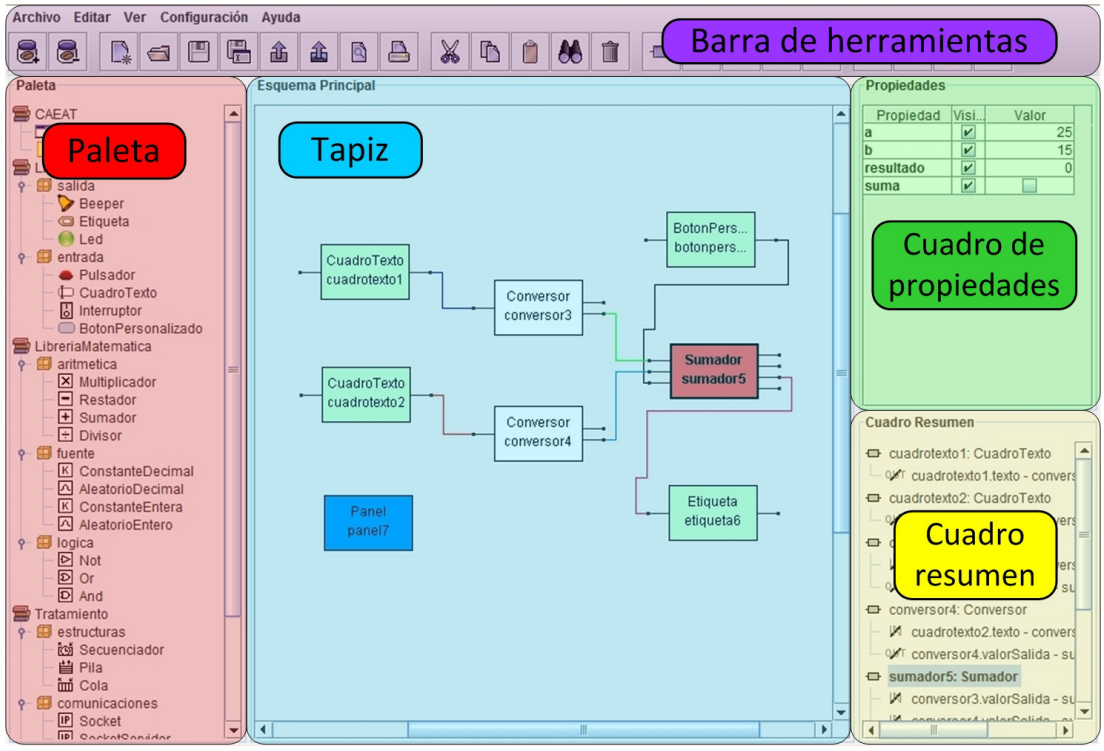

Figura 2.5: Desglose por partes de la interfaz de usuario de CAEAT

2.2.1. Paleta

En la parte izquierda de la interfaz se encuentra la paleta de componentes. En ella se muestran, en forma de árbol, todos los componentes software disponibles en la plataforma en un momento determinado. Al pulsar sobre uno de ellos, es posible insertarlo en el tapiz de CAEAT, donde recibirá una representación gráfica como la mostrada en el apartado anterior.

Por defecto, la paleta se encuentra vacía a excepción de algunos componentes básicos de la plataforma. Es posible poblarla añadiendo librerías de componentes contenidas en archivos de formato *jar*. El formato y las características de estas librerías se discuten en apartados posteriores.

Como cualquier menú de selección en forma de árbol, la paleta permite expandir y colapsar los nodos de los cuales "cuelgan" las listas de componentes de cada librería. Asimismo, la paleta muestra cada componente de una manera amigable para el usuario, incluyendo un icono que se habrá proporcionado dentro del archivo *jar* de la librería y una pequeña descripción a modo de *tooltip* al colocar el puntero sobre él.

2.2.2. Tapiz

El tapiz constituye el área principal de trabajo de CAEAT, por lo cual se le dota de la mayor superficie dentro de la interfaz de usuario. Consiste en un panel con barras de desplazamiento vertical y horizontal. En el tapiz se muestra el esquema actual, con sus correspondientes componentes y conexiones. Las acciones más comunes a realizar sobre el tapiz son las siguientes:

- Agregación y borrado de componentes.
- Creación y borrado de conexiones entre componentes.
- Encapsulación de componentes para formar un componente compuesto que los englobe y sea visto desde fuera como una "caja negra" más. En este caso es necesario seleccionar cuáles de las patillas de los componentes englobados serán visibles desde el exterior de la nueva "caja".
- Modificación de los atributos visuales de los componentes: su posición, su color y su nombre. Modificación asimismo de los colores de las conexiones.
- Copiado, cortado y pegado de los componentes.
- Ocultación (pero no eliminación) de las conexiones entre componentes, en caso de que la densidad de éstas no permita trabajar de manera cómoda con el esquema.
- Navegación por la jerarquía de componentes compuestos del esquema. Si el tapiz contiene un componente formado por diversos sub-componentes, basta realizar un doble *click* sobre él para visualizar el sub-esquema encapsulado por dicho componente compuesto.
- Inserción de notas recordatorias a modo de *post-it*. Estas notas no tienen influencia sobre el funcionamiento de la agregación de componentes.

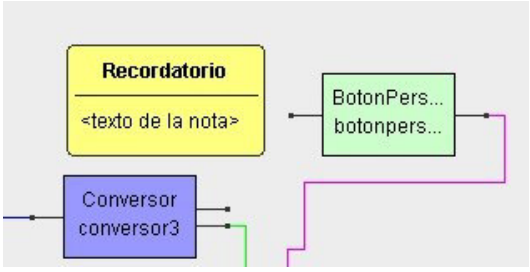

Figura 2.6: Inserción de notas recordatorias sobre los esquemas del tapiz

- Acceso a las propiedades de un componente (o conexión). Al pulsar con el botón izquierdo del ratón sobre un componente (o conexión), se permite la edición de sus propiedades mediante una tabla insertada en el cuadro de propiedades (véase apartado inmediatamente posterior a éste). De manera similar, al realizar doble *click* sobre un componente sencillo (es decir, que no contenga dentro de sí más componentes recursivamente), también es posible la edición visual de sus propiedades mediante su interfaz *Customizer* (concepto que se expone en el capítulo 2.3).

Se debe destacar que la lógica implementada alrededor del tapiz gestiona adecuadamente la disposición de los elementos sobre el mismo. No es posible la superposición de componentes, del mismo modo que el trazado de las conexiones es calculado de manera que no interseque ninguna "caja" ó patilla.

El título del tapiz muestra información sobre el esquema actualmente bajo visualización. Si se está viendo el esquema principal mostrará "Esquema Principal", pero en el momento en que se navegue por el interior de un componente compuesto mostrará toda la jerarquía de componentes que hay por encima de él (por ejemplo, mostrará "Esquema Principal -> Red 1 -> Red 2 -> Red 3" si se está visualizando la "Red 3" y ésta tiene por encima a la "Red 2" y la "Red 1").

2.2.3. Cuadro de propiedades

En esta tabla se muestran las propiedades del elemento que se encuentra actualmente bajo selección en el tapiz. Su contenido cambia sustancialmente si dicho elemento se trata de un componente o una conexión.

- **Componente**: se muestran en forma de tabla las propiedades visibles de cada componente. En la mayoría de casos, las propiedades de un componente serán tipos de datos primitivos del lenguaje de programación *Java*, aunque también pueden consistir en otros tipos de objetos. La naturaleza de las propiedades de un componente es explicada con más detalle en el siguiente apartado.

  Dado que cada propiedad de un componente va asociada a una patilla de la "caja negra" que lo representa, es posible elegir en esta tabla si se desea que la caja muestre la patilla o no. Además, si la propiedad es editable en formato textual, se permite la inserción del valor nuevo en la tercera columna de la tabla.

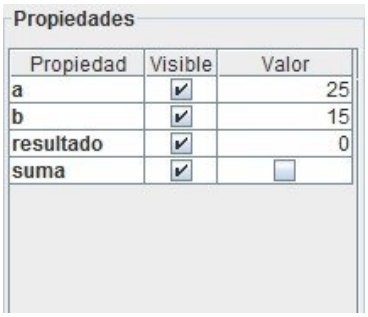

Figura 2.7: Cuadro de propiedades cuando se carga un componente

- **Conexión**: se muestra una tabla con información sobre el origen y el destino de la conexión; tanto el nombre de los componentes que hay en ambos extremos de la conexión como las patillas de la cual parte y a la cual llega la conexión. Adicionalmente se proporciona una casilla que abre un selector de colores al clicar sobre ella y facilita el cambio de color de la conexión.

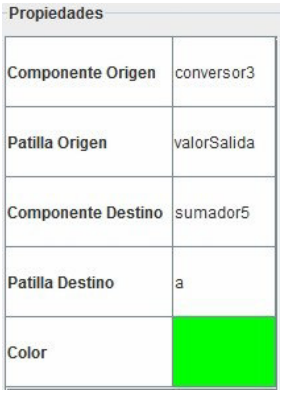

Figura 2.8: Cuadro de propiedades cuando se carga una conexión

2.2.4. Cuadro resumen

Este cuadro contiene una representación en forma de árbol de todos los elementos contenidos actualmente en el tapiz. Se muestran los nombres de todos los componentes y, colgando de ellos, las conexiones entrantes y/o salientes que éstos contengan. Se utilizan iconos diferentes para identificar rápidamente conexiones salientes y entrantes. Desde este cuadro es posible ocultar o mostrar conexiones del tapiz automáticamente realizando doble clic sobre su nombre. También se permite abrir un formulario mediante el cual localizar elementos del tapiz mediante su nombre o una porción del mismo, funcionalidad muy útil si se está trabajando con un esquema denso.

2.2.5. Barra de herramientas

Como cualquier otra plataforma software similar, CAEAT incluye una barra de herramientas con menús desplegables que permiten activar todas las funciones de la herramienta, así como un espacio con iconos de acceso directo a las funcionalidades más habituales.

A continuación se muestra un listado rápido con las acciones que es posible llevar a cabo a través de los diferentes menús. Sirva dicho listado a modo de resumen de las distintas operaciones que era posible realizar en la plataforma CAEAT antes del inicio del presente proyecto.

- **Archivo**: ofrece las acciones típicas de manejo de los archivos. Permite guardar y cargar esquemas guardados, iniciar un nuevo esquema y añadir una librería a la paleta. Ofrece opciones para el manejo de imágenes, ya que es posible imprimir el contenido del tapiz: *Exportar imagen*, *Configurar página*, *Vista previa* e *Imprimir*. También ofrece un espacio en el que almacena accesos directos a los últimos esquemas utilizados. Por último, permite exportar el esquema actual a un archivo autoejecutable, característica que se explica con más detalle en un apartado posterior.
- **Editar**: ofrece opciones relacionadas con el manejo de los componentes que hay sobre el tapiz. Es posible eliminar, cortar, copiar y pegar componentes, buscar elementos según su nombre y encapsular varios componentes dentro de un componente compuesto.
- **Ver**: ofrece opciones relacionadas con la visualización del tapiz. Es posible navegar por la jerarquía de anidamiento de los componentes compuestos mediante los botones *Restaurar esquema principal*, *Restaurar esquema anterior* y *Mostrar esquema*. También se puede mostrar una rejilla sobre el tapiz y, en caso de ser ésta visible se puede ordenar que los componentes se alineen sobre ella. Es posible asimismo dar la orden de mostrar todas las conexiones que se encuentren ocultas y mostrar u ocultar las notas recordatorias. Por último, se puede mostrar el asistente de creación de interfaces gráficas, funcionalidad que se explica en un apartado posterior.

- **Configuración**: ofrece algunas opciones que no encajan en los menús anteriores. Es posible seleccionar la carpeta en la que se guardarán las librerías (es decir, los archivos *jar*) nuevas que se añadan. Se permite editar el color por defecto que CAEAT asigna a todos los componentes de una misma librería. Si hay elementos de la interfaz gráfica sobre el tapiz, se puede cambiar el panel principal que los mostrará por pantalla. Por último, es posible activar, desactivar y cambiar las opciones de *logging* (las trazas en forma textual que el programa saca por su salida estándar para informar de errores y otras circunstancias).
- **Ayuda**: contiene un botón llamado *Acerca de CAEAT* que ofrece información sobre la versión y los autores de la plataforma.

Figura 2.9: Menús desplegables ofrecidos por CAEAT

La botonera horizontal bajo la barra de menús contiene accesos rápidos a algunos de los comandos más frecuentes de entre los que se han expuesto.

2.3. FILOSOFÍA DE DISEÑO SOFTWARE

A la hora de abordar el diseño y codificación de una plataforma software como la aquí presentada, suelen surgir problemáticas comunes y ampliamente documentadas en el desarrollo de plataformas similares en el pasado. Para solucionar estas situaciones recurrentes surgieron los patrones de diseño. En ingeniería del software, un patrón de diseño es una solución reutilizable y aplicable a un problema concreto y recurrente dentro de un determinado contexto. Se presentan en este capítulo los dos patrones de diseño fundamentales que implementa la plataforma CAEAT.

2.3.1. Patrón de diseño *Model – View – Controller*

La plataforma CAEAT ha sido enteramente creada siguiendo el patrón de diseño software *Model – View – Controller* (MVC). El presente proyecto ha respetado este diseño en todas las ampliaciones y modificaciones que se han realizado.

MVC es un patrón que surge en el año 1979 en el contexto del lenguaje de programación *Smalltalk*. Este patrón apuesta por separar y desacoplar las tres partes fundamentales que forman una aplicación: la interfaz de usuario, los datos de programa y la lógica de control que se encarga de las operaciones.

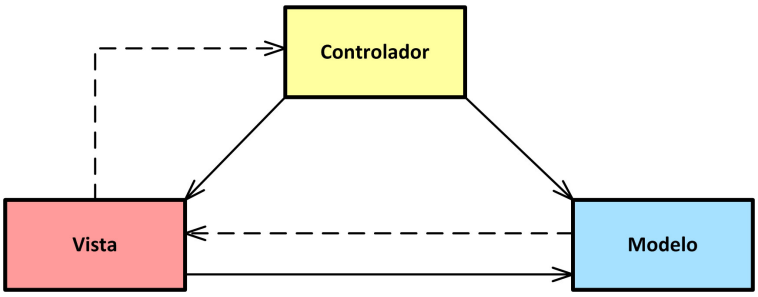

Figura 2.10: Representación del patrón de diseño *Model – View – Controller*

- **Modelo**: parte encargada de almacenar y administrar los datos lógicos de la aplicación. Usualmente tiene como finalidad servir de abstracción de algún proceso en el mundo real. Contiene todas las funcionalidades necesarias para controlar y garantizar la integridad del sistema y de los datos de proceso.
- **Vista**: representación visual del modelo en pantalla. Engloba todas las partes que usualmente suelen denominarse "interfaz de usuario". Puesto que está asociada al modelo, cuando hay una modificación en dicho modelo, ésta debe actualizarse adecuadamente para reflejar los cambios.
- **Controlador**: parte encargada de las funciones de control, usualmente como respuesta a eventos y acciones iniciadas por el usuario. Invoca peticiones a la vista y al modelo para gestionar estos eventos adecuadamente y llevar a cabo los procesos necesarios.

Una de las grandes ventajas que presenta este patrón es el desacoplamiento total entre la vista y modelo. Gracias a él, se pueden llevar a cabo acciones como la remodelación completa de la interfaz de usuario de una aplicación sin afectar al comportamiento lógico del programa.

En el esquema presentado, las líneas sólidas indican asociación directa. Esto significa que las llamadas y peticiones desde el bloque origen se realizan directamente sobre el bloque destino (por ejemplo, el controlador puede llamar a funciones alojadas en la vista y en el modelo). En cambio, las líneas punteadas indican asociación indirecta: estas peticiones se realizan indirectamente mediante el lanzamiento de eventos o mediante la implementación de otro patrón de diseño ampliamente usado: *Observer – Observable*.

2.3.2. Patrón de diseño *Observer – Observable*

Como se puede observar en la figura que describe el patrón MVC, la vista conoce completamente a su modelo, y puede realizar llamadas directas sobre él. Sin embargo, el recíproco no es cierto. El modelo no conoce nada de su respectiva vista, y por lo tanto es necesario un mecanismo de notificación indirecta para que el modelo pueda pedir a la vista que se actualice para reflejar un cambio en los datos del programa. Se utiliza en este caso el patrón *Observer – Observable*, que define los dos roles que lleva en su nombre.

- ***Observer***: espera la notificación de cambios por parte de uno de los objetos observados para actualizar su estado. Para que un objeto actúe como *Observer*, únicamente se debe implementar la interfaz del mismo nombre del lenguaje de programación *Java*. La notificación llega a través de la llamada a un método *update* que realizarán los objetos observados cuando sea necesario. En el caso de CAEAT, los objetos que forman la Vista de la aplicación esperan notificación por parte del Modelo para poder actualizarse.
- ***Observable***: objetos observados que notifican sus cambios a los observadores cuando es necesario hacerlo. Para que un objeto actúe como *Observable*, se debe heredar de la clase del mismo nombre del lenguaje de programación *Java*. Esta clase proporciona métodos para

añadir objetos observadores a una lista, así como el método llamado *notifyObservers*, que provoca la llamada del método de actualización *update* de todos los observadores de la lista. En el caso de CAEAT, los objetos del Modelo son los observados (y por ende heredan de la clase `Observable`), y se encargan de notificar sus cambios a la Vista.

El presente proyecto ha introducido la utilización de otros patrones de diseño en la solución de problemáticas muy concretas que se han presentado durante su desarrollo. Estos patrones son detallados y ejemplificados en el anexo E.

2.4. JAVA BEANS

Hasta el momento se ha hablado de que los componentes manejados por CAEAT encapsulan piezas simples de software que realizan funciones muy sencillas. También se ha argumentado que estos fragmentos de software dejan expuestas una serie de propiedades con las cuales se puede interactuar y que CAEAT representa en forma de patillas que es posible interconectar entre sí. Este apartado expone los detalles de implementación en el lenguaje de programación *Java* de estos mecanismos.

La "pieza de software" que hay por debajo de uno de los componentes de CAEAT no es más que una clase del lenguaje de programación *Java*. Esta clase (como todas las clases *Java*) define una serie de atributos, la mayoría de ellos privados. Para poder manejarlos, la clase define una serie de métodos de tipo "set" y "get", que permiten establecer y consultar el valor de los atributos respectivamente. Estos atributos son los que quedan expuestos al exterior en forma de patillas. Un cambio en un atributo puede provocar que la clase ejecute un código que realice alguna pequeña función que a su vez afecte al valor de otro atributo. Se consigue así el comportamiento de "componente software que realiza una tarea simple" que se describió con anterioridad.

Este comportamiento se consigue mediante el uso de las herramientas definidas por el estándar *JavaBeans*. Este estándar consiste en un modelo de programación (un conjunto de API's) creado por *Sun Microsystems* junto con las primeras distribuciones del lenguaje de programación *Java* en 1996. El concepto de "bean", definido a continuación, se adapta muy bien a un entorno como el que propone la herramienta CAEAT.

2.4.1. Definición de *bean*

La especificación de *JavaBeans* define a los *beans* como "componentes de software reutilizables que se puedan manipular visualmente en una herramienta de construcción", una definición que encaja a la perfección con lo que se pretende realizar con la plataforma CAEAT. Además, según la especificación, un *bean* debe cumplir 5 características esenciales. A continuación se exponen estas 5 características y la metodología empleada en CAEAT para su aplicación.

1. **Introspección**: una herramienta software externa debe ser capaz de determinar cómo funciona el *bean* mediante el análisis del código fuente de su clase. En el lenguaje de programación *Java* esto se consigue con la ayuda de las clases `BeanDescriptor` e `Introspector`, además del seguimiento de unos convenios semánticos definidos por las especificaciones de *JavaBeans* a la hora de implementar el *bean*. Como se detalla en el ejemplo mostrado en el anexo A, en el caso de la plataforma CAEAT se han seguido estos patrones definidos por el estándar.
2. **Customización**: un *bean* debe permitir la personalización de sus propiedades, su aspecto y/o su comportamiento. En la plataforma CAEAT cada *bean* va acompañado de una clase `Customizer`, consistente en un cuadro de diálogo que permite seleccionar de una manera gráfica y agradable para el usuario los diferentes parámetros relativos a dicho *bean*. Es

posible acceder al `Customizer` de cada bean realizado doble *clic* sobre el componente correspondiente en el tapiz de la plataforma.

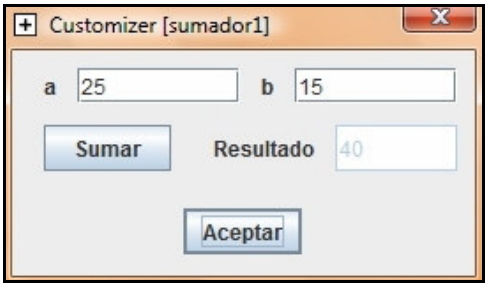

Figura 2.11: *Customizer* ofrecido por el *bean* Sumador

3. **Eventos**: el paradigma de la comunicación entre *beans* se basa en el lanzamiento de eventos del lenguaje de programación *Java*. En la plataforma CAEAT el tipo de evento empleado es `PropertyChangeEvent`, que es lanzado cada vez que una de las propiedades del *bean* es modificada. Cuando dos componentes del tapiz son conectados mediante una conexión gráfica, en realidad se están registrando cada uno como *listener* de los eventos lanzados por el otro. De esta manera, mediante cadenas de eventos, se consigue simular el "viaje" de una información (y sus posibles transformadas) a través de las conexiones creadas por el usuario cada vez que éste modifica una propiedad en un componente.

4. **Propiedades**: según el estándar, un *bean* debe ofrecer una serie de propiedades customizables bien definidas y obtenibles mediante introspección. La especificación recomienda definir métodos del tipo *get<nombrePropiedad>* y *set<nombrePropiedad>* para consultar y modificar, respectivamente, el valor de las propiedades. Como ya se ha avanzado anteriormente, el diseño de la plataforma CAEAT ha seguido las recomendaciones del estándar y las propiedades de cada bean son analizadas y representadas en forma de patillas que salen de las "cajas negras" que representan a los componentes.

5. **Persistencia**: un bean debe proporcionar soporte para salvar su estado de manera permanente y poder restaurarlo a posteriori. En la plataforma CAEAT se ha obligado a que los beans implementen la interfaz `Serializable` del lenguaje de programación *Java*, cuya utilidad es permitir que un objeto pueda ser convertido a una ristra de *bytes* apta para ser almacenada en un fichero o transmitida por la red. Los detalles del proceso de serialización se han detallado anteriormente en el apartado "1.2.3.3. Serialización de objetos Java".

La especificación completa y oficial de *JavaBeans* publicada por *Sun Microsystems* en agosto de 1997 es actualmente propiedad de *Oracle* y se puede consultar gratuitamente en la referencia [**5**].

2.4.2. Estructura de clases de un *bean*

La implementación de un *bean* utilizable por la herramienta de edición CAEAT implica el desarrollo de tres clases *Java* bien diferenciadas, cuya naturaleza y finalidades se enumeran a continuación:

- ***Bean***: se trata de la clase principal en la cual residen las propiedades del *bean*. Esta clase proporciona los métodos *getter* y *setter* necesarios para consultar y modificar (respectivamente) las propiedades, y en ella se lleva a cabo el procesado ofrecido por el *bean*. Adicionalmente admite la agregación de *listeners* a los cuales se notificarán los cambios en el valor de las propiedades, característica utilizada por CAEAT para la representación visual de los *beans* como componentes interconectados entre sí.

  El extracto de código siguiente ejemplifica el formato seguido por un *bean* para la declaración de una de sus propiedades junto con sus métodos *getter* y *setter*:

```
        private int a;
        private PropertyChangeSupport cambios;

        public int getA() {
                return a;
        }

        public void setA(int a) {
                int aAnt = this.a;
                if (aAnt != a) {
                        this.a = a;
                        cambios.firePropertyChange("a", aAnt, a);
                }
        }
```

Código 2.1: Métodos *get* y *set* correspondientes a una propiedad de un *bean* sencillo

Nótese asimismo el uso de un objeto de tipo `PropertyChangeSupport` que almacena una lista con los *listeners* o entidades interesadas en la recepción de los eventos generados por parte de este bean, y su utilización para el lanzamiento de un evento de cambio de propiedad cuando ésta es modificada.

- ***BeanInfo***: clase utilizada por la Máquina Virtual de *Java* para realizar introspección sobre el *bean*, es decir, para determinar los métodos y propiedades que éste ofrece mediante el análisis de su código fuente. Siguiendo una semántica bien definida por el estándar *JavaBeans*, en esta clase es posible especificar los nombres de los métodos ofrecidos por el *bean*, establecer las propiedades que éste contiene y determinar cuáles son de lectura, escritura o ambas, ofrecer un icono descriptivo del bean, etc.
- ***Customizer***: la generación de esta clase es opcional según el estándar de *JavaBeans*. Se trata de una clase que implementa una interfaz gráfica (que puede ser más o menos compleja a elección del programador) que permite la edición de las propiedades del *bean* de una manera intuitiva y agradable para el usuario. En la plataforma CAEAT, el *Customizer* de los *beans* es accesible realizando doble *clic* sobre el componente que los representa en el tapiz.

En el anexo A se muestra y comenta en profundidad el código fuente completo de uno de los *beans* disponibles en la librería matemática de CAEAT: el sumador de dos números enteros.

2.5. LIBRERÍAS DE COMPONENTES IMPLEMENTADAS

En el momento del inicio del presente proyecto, junto con la plataforma CAEAT, se encontraban programadas tres librerías de componentes que permitían generar agregaciones de funcionalidades muy diversas. Este apartado realiza una breve descripción de cada uno de los componentes disponibles en CAEAT.

Pese a que el presente proyecto no tiene como objetivo principal la generación de nuevas librerías de componentes, sí se ha actuado sobre los componentes ya existentes previamente al inicio de éste, para hacerlos compatibles con las nuevas características de despliegue en red introducidas en la herramienta y detalladas en los capítulos subsiguientes de este documento.

2.5.1. Librería Matemática

Contiene *beans* que realizan las funciones aritméticas básicas, así como algunas funciones de lógica booleana (verdadero / falso). También contiene generadores de números, tanto aleatorios como deterministas, y tanto de tipo entero como decimal.

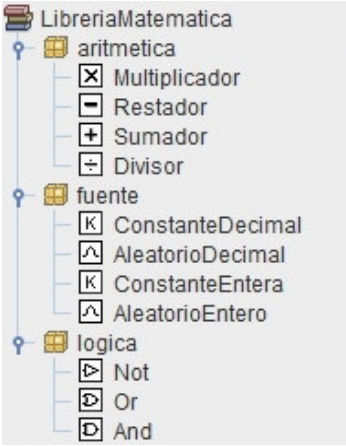

Figura 2.12: Aspecto de la Librería Matemática en la paleta de CAEAT

- **Sumador**: realiza la suma de dos números enteros en el momento en que se le indica a través de una orden explícita (que se le proporciona mediante el paso a *true* de un valor de tipo *boolean*). Devuelve el resultado en un atributo de sólo lectura.
- **Restador**: idéntico comportamiento al del sumador, realizando en este caso la resta de dos números enteros.
- **Multiplicador**: idéntico comportamiento al del sumador, realizando en este caso la multiplicación de dos números enteros.
- **Divisor**: realiza la división de dos números enteros y devuelve el resultado en dos atributos distintos representando el resultado entero y el resto o módulo.
- **ConstanteEntera**: genera un valor de tipo entero constante y determinista bajo petición del usuario (nuevamente, mediante el paso a *true* de un valor de tipo *boolean*).
- **ConstanteDecimal**: idéntico comportamiento al *bean* anterior, pero generando en este caso un valor de tipo decimal (coma flotante).
- **AleatorioEntero**: genera y devuelve un valor aleatorio entero bajo petición del usuario. Es posible definir un rango de valores máximo y mínimo entre los cuales estará comprendido el valor devuelto, o tomar por defecto todo el rango de valores enteros representables en el lenguaje de programación *Java*.
- **AleatorioDecimal**: idéntico comportamiento al del *bean* anterior, pero devolviendo en este caso un valor decimal (coma flotante).
- **And**: realiza a petición del usuario la operación lógica “AND” con dos valores *boolean* (verdadero / falso) que le son proporcionados, y devuelve el resultado en un atributo de sólo lectura.
- **Or**: idéntico comportamiento al del *bean* And, pero realizando la operación lógica “OR” entre los valores *boolean* proporcionados.
- **Not**: realiza la negación de un valor lógico proporcionado, almacenando el resultado en un atributo *boolean* de sólo lectura.

2.5.2. Librería de Tratamiento

Contiene *beans* relacionados con el tratamiento, conversión y almacenamiento de datos, así como con la comunicación entre componentes u otras entidades.

Figura 2.13: Aspecto de la Librería de Tratamiento en la paleta de CAEAT

- **Pila**: almacena elementos y objetos de cualquier tipo siguiendo una estructura LIFO (*Last In, First Out*). Permite definir el número máximo de elementos de la estructura, apilar y desapilar elementos, consultar el valor del elemento que hay en la "cima" de la pila, vaciar la estructura y consultar la longitud de la misma.
- **Cola**: estructura de datos muy similar a la pila pero que implementa la disciplina de una cola FIFO (*First In, First Out*). Permite definir el número máximo de elementos, consultar el valor del primer y el último elemento, encolar y desencolar elementos, vaciar la estructura y consultar la longitud actual de ésta.
- **Secuenciador**: estructura de datos idéntica a la cola pero con la particularidad de que desencola los elementos de manera automática y periódica según un periodo de tiempo definido por el usuario en milisegundos.
- **Conversor**: permite cambiar el tipo de datos de un objeto (realizar un *cast*), seleccionando el tipo de valor de salida de entre todos los tipos de datos nativos del lenguaje de programación *Java*. La conversión puede realizarse de manera automática cuando el valor de entrada es modificado o únicamente bajo demanda del usuario.
- **Comparador**: compara dos valores enteros *a* y *b* e indica si *a* es mayor, menor o igual que *b* mediante la activación (puesta a "verdadero") de la correspondiente salida de tipo *boolean*. La comparación puede realizarse automáticamente cuando se detecta un cambio en los atributos de entrada o únicamente bajo petición del usuario.
- **Temporizador**: genera un pulso cada cierto tiempo (seleccionable por el usuario en milisegundos). Dicho pulso no es más que la puesta a *true* de su salida de tipo *boolean* y de sólo lectura. Permite escoger que este pulso sea instantáneo o que se mantenga a *true* durante el 50% del periodo de espera entre pulsos.
- **Inversor**: invierte automáticamente el valor de tipo *boolean* que se le proporciona a la entrada. *Bean* muy similar al Not de la Librería Matemática.
- **Selector**: admite como entrada un valor lógico (*boolean*). Redirige este valor a una de sus cuatro salidas, seleccionable por el usuario. Es un componente útil para redireccionar eventos adecuadamente entre el resto de componentes.

- **SelectorEntero**: comportamiento idéntico al *bean* Selector, con la diferencia de que el parámetro admitido como entrada, y por lo tanto redireccionado por una de las salidas, es un valor de tipo entero.
- **Socket**: *bean* que encapsula los detalles de manejo de un objeto de la clase `Socket` del lenguaje de programación *Java*. Permite abrir una conexión con un *host* indicando su dirección IP y el puerto adecuado. Si la conexión es exitosa, permite enviar y recibir mensajes de texto a través del *Socket*.
- **SocketServidor**: de manera similar a Socket, encapsula los detalles de manejo de la clase `ServerSocket` del lenguaje de programación *Java*. Permite elegir un puerto TCP en el cual escuchar conexiones entrantes a la máquina desde la que es ejecutado. Proporciona como valor de salida los mensajes recibidos a través de las conexiones entrantes.

2.5.3. Librería Gráfica

Contiene los componentes que incluyen interfaz gráfica, y que por tanto podrán ser usados para dotar de interfaz de usuario a las agregaciones de componentes creadas mediante el asistente que se describe en el apartado siguiente.

Figura 2.14: Aspecto de la Librería Gráfica en la paleta de CAEAT

- **Pulsador**: implementa la funcionalidad de un botón “normalmente abierto”; es decir, cambia su valor de salida a “verdadero” al pulsarse y regresa automáticamente a “falso” al dejar de pulsarse.
- **Interruptor**: botón de estado permanente; es decir, no regresa al estado “falso” al dejar de pulsarse, sino cuando el usuario decide que debe hacerlo (pulsándolo de nuevo).
- **BotonPersonalizado**: botón de comportamiento idéntico al pulsador, con la diferencia de que permite la customización de su aspecto externo. Concretamente, permite elegir la leyenda que mostrará el botón en la interfaz de usuario y su color de fondo.
- **CuadroTexto**: implementa la funcionalidad de un cuadro en el cual es posible introducir texto para usar éste como parámetro de entrada a otros componentes.
- **Beeper**: *bean* que reproduce un sonido en los altavoces de la máquina en la que se ejecuta cuando su propiedad de tipo *boolean* de entrada pasa a *true*. Además, su representación gráfica en pantalla, consistente en una campana, sufre una animación para que la alarma sea más detectable.
- **Led**: *bean* que simula ser un semáforo con tres estados conmutables mediante su atributo de entrada. Para cada estado, la representación gráfica del bean es distinta: una luz verde, ámbar y roja.
- **Etiqueta**: cuadro que muestra texto no editable (únicamente de lectura) en la interfaz de usuario de la agregación.

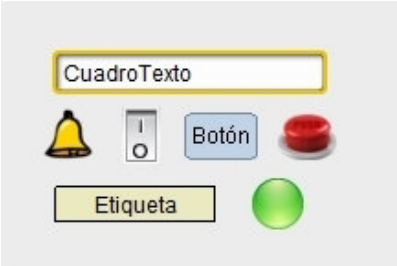

Figura 2.15: Aspecto final de los componentes de la Librería Gráfica en la interfaz

2.5.4. Librería CAEAT

Aunque no constituyen una librería externa ya que no se distribuyen en un archivo *jar* externo, se hace mención a continuación de dos componentes fundamentales en la construcción de agregaciones de componentes que han sido embedidos dentro de la propia plataforma.

Figura 2.16: Aspecto de la librería de componentes básicos embedida en CAEAT

- **Nota**: componente que permite insertar notas recordatorias, tanto sobre el tapiz de CAEAT durante la construcción de una agregación como en su interfaz gráfica generada mediante los elementos de la Librería Gráfica.
- **Panel**: componente gráfico contenedor de otros componentes gráficos. Tal y como se detalla en el apartado siguiente, es posible agregar a este componente los *beans* que dispongan de interfaz gráfica para que ésta sea mostrada como la interfaz hombre-máquina de la agregación que se ha construido.

2.6. DISEÑO DE INTERFACES GRÁFICAS

Hasta el momento se ha expuesto el mecanismo usado por CAEAT para representar a los *beans* (fragmentos sencillos de software) como componentes o cajas negras que es posible interconectar entre ellas a fin de que colaboren en la realización de una función compleja. Aunque se ha expuesto que es posible controlar gráficamente las propiedades de los componentes mediante el *Customizer* asociado a cada uno de ellos, el objetivo de la plataforma CAEAT es ofrecer interfaces gráficas mucho más intuitivas para conseguir tal fin.

Con ese objetivo nacieron los componentes gráficos, detallados en el apartado anterior, y el Asistente de Creación de Interfaces Gráficas, que se detalla a continuación. Mediante la creación de una interfaz gráfica que permita controlar los parámetros de una agregación a la vez que visualizar sus salidas, es posible desacoplar el proceso de creación de redes de su proceso de utilización o ejecución. Tal y como se detalla en el apartado 2.8 (que versa sobre las agregaciones autoejecutables), disponer de una interfaz hombre-máquina independiente a la plataforma CAEAT es fundamental para la utilización de las agregaciones de manera autónoma.

Para poder usar el asistente de creación de interfaces gráficas es necesario que la agregación contenga componentes gráficos (uno como mínimo) y un Panel contenedor. El diálogo *Customizer* del Panel gráfico permite seleccionar qué subconjunto de los elementos gráficos alojados en el tapiz deberán estar contenidos en dicho Panel:

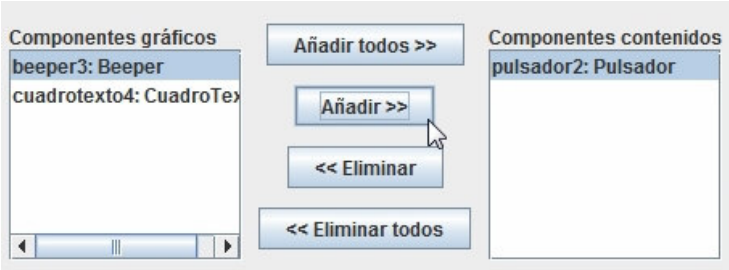

Figura 2.17: Selección de los elementos gráficos contenidos en un Panel mediante su *Customizer*

El asistente de creación de interfaces gráficas consiste en un panel que representa de manera simbólica la disposición de los elementos gráficos. Es posible redimensionar el panel y mover los elementos sobre él, así como redimensionarlos como se desee. Si hay más de un panel en el tapiz, también es posible elegir cuál será el panel principal de la agregación. Si se aceptan los cambios, el panel diseñado pasará a ser la interfaz actual de la agregación. Es posible previsualizar dicha interfaz desde el tapiz de CAEAT, para comprobar si su aspecto final es el deseado.

En la siguiente figura puede apreciarse el aspecto del asistente gráfico con tres elementos bajo edición, así como el aspecto final de la interfaz gráfica al finalizar dicha edición.

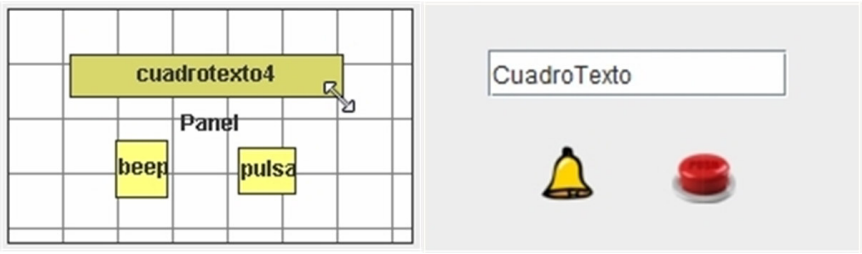

Figura 2.18: Creación de una interfaz gráfica sencilla mediante el Asistente

2.7. GUARDADO DE ESQUEMAS

La herramienta CAEAT permite guardar en un fichero el estado de una agregación. Esto resulta útil para mover agregaciones entre máquinas diferentes, o para poder salvar la red si se debe interrumpir su proceso de construcción y se debe reanudar en el futuro. Se ha elegido la extensión *aec* para denotar los ficheros generados y comprensibles por la plataforma CAEAT.

Como se ha comentado en un apartado anterior, todos los *beans* deben ofrecer soporte para persistencia, es decir, para poder salvar su estado y restaurarlo a posteriori. En CAEAT se obliga a todos los *beans* a implementar la interfaz `Serializable`, que ofrece mecanismos para transformar un objeto en una ristra de bytes que puede ser fácilmente escrita en un fichero usando clases proporcionadas por la especificación del proceso de serialización (consultar la referencia [**2**] para más información).

Garantizado el almacenaje sencillo de los *beans*, cabe preguntarse por el formato de guardado de la meta-información que rodea a una agregación: conexiones, encapsulaciones, posición de los elementos sobre el tapiz, color de los elementos, contenido de los paneles, etc. Durante el diseño inicial de CAEAT se decidió encapsular esta información en un documento XML que se serializa en el mismo fichero *aec* junto con los beans que forman la red.

Para la creación, manejo y lectura rápida de documentos XML se decidió usar la librería de código abierto JDOM, que proporciona un modelo de programación (una serie de clases del lenguaje de programación *Java*) que permiten llevar a cabo estas operaciones de forma sencilla. Las especificaciones completas de dicha librería pueden consultarse en la referencia [**6**].

A continuación se presenta a modo de ejemplo el formato escogido para definir una agregación sencilla, formada únicamente por una ConstanteEntera y un Sumador conectados entre sí:

```
<Esquema Version="2">
  <Componentes>
      <Componente NombreInstancia="constanteentera1" Color="-52429"
      NombreClase="sener.matematica.fuente.ConstanteEntera">
      <X>336</X>
      <Y>347</Y>
    </Componente>
      <Componente NombreInstancia="sumador2" Color="-52429"
      NombreClase="sener.matematica.aritmetica.Sumador">
      <X>478</X>
      <Y>472</Y>
    </Componente>
  </Componentes>
  <Conexiones>
    <Conexion Color="-16776961">
      <Origen Patilla="valor" Componente="constanteentera1" />
      <Destino Patilla="a" Componente="sumador2" />
    </Conexion>
  </Conexiones>
  <Paneles />
  <Notas />
</Esquema>
```

Código 2.2: Ejemplo de documento XML descriptor de una red de componentes sencilla

Cabe destacar que para cada componente se especifica toda la meta-información necesaria para reconstruir la red, como el color en formato entero y las coordenadas X e Y del tapiz en las cuales se debe insertar. Se detallan también las conexiones entre componentes, especificando origen y destino. También se facilita información sobre las posibles notas recordatorias insertadas en el esquema, así como los paneles que pudiese contener y los elementos gráficos contenidos en cada panel (el ejemplo mostrado no contiene notas ni paneles). Si un componente es compuesto (contiene otros componentes en su interior) se detallan recursivamente los datos de cada uno de ellos en el documento XML.

En el momento de cargar el esquema, se de-serializan los *beans*; es decir, se reconstruyen los objetos originales a partir de los *bytes* escritos en el fichero *aec*. Con la ayuda del documento XML, es posible reconstruir toda la agregación, ya que contiene toda la información necesaria para:

- Recolocar cada componente adecuadamente sobre el tapiz, con el mismo nombre y el mismo color con el que fue guardado.
- Reconstruir los componentes compuestos, si los hay.
- Reconstruir las conexiones entre componentes, así como su color.
- Restaurar las interfaces gráficas (inicializar los paneles adecuadamente, si los hubiese).
- Restaurar las notas recordatorias que el esquema pudiese contener.

2.8. AGREGACIONES AUTOEJECUTABLES

Tal y como se ha apuntado en un apartado anterior, los objetivos de la plataforma de procesado que se ha denominado *SeNetComponents* van más allá de la herramienta de ensamblaje de componentes CAEAT. El alcance final de *SeNetComponents* es el despliegue, gestión y edición de agregaciones de componentes de manera autónoma en entornos muy diversos. Bajo esta visión, la herramienta CAEAT juega el papel de plataforma generadora de las agregaciones que más adelante podrán ser ejecutadas de manera independiente en entornos que no dispongan de CAEAT.

Este enfoque implicaba que CAEAT dispusiera de un mecanismo para guardar las agregaciones de manera que éstas pudiesen ser ejecutadas y utilizadas sin necesidad de CAEAT. Se diseñó así la exportación de agregaciones a archivos *jar* autoejecutables. De esta manera, una agregación que se da por finalizada puede no sólo guardarse en un archivo *aec* para poder ser abierta a posteriori desde CAEAT, sino también en un archivo *jar* que podrá ser ejecutado mediante cualquier Máquina Virtual de *Java*.

El proceso de encapsulación de una agregación en un archivo *jar* comprende los siguientes pasos:

1. Creación del archivo *jar* que contendrá todos los recursos necesarios para la ejecución autónoma.
2. Generación del archivo *aec* de la misma manera que si se realizase un simple guardado, y almacenaje de éste en el archivo *jar*.
3. Exportación al archivo *jar* de las clases de CAEAT que pueden ser reutilizadas. La mayoría de estas clases corresponden al Modelo de la plataforma. Dado que no habrá interfaz de CAEAT, todas las clases relativas a la Vista son inservibles en este entorno. Sí serán útiles algunas clases del Controlador, así como la clase principal que permitirá ejecutar el conjunto como una aplicación *Java* independiente.
4. Inclusión en el archivo *jar* de las librerías (archivos *jar*) a las cuales pertenecen los *beans* implicados en la agregación que se está exportando, así como de las librerías externas utilizadas por CAEAT que sean necesarias (como por ejemplo la librería de manejo de documentos XML, “JDOM”).
5. Inclusión del correspondiente archivo de manifiesto *Manifest.mf*, necesario para que la Máquina Virtual de *Java* tenga conocimiento de cómo se debe ejecutar el archivo *jar*. Un archivo de manifiesto no es más que un archivo de texto que sigue una semántica muy concreta, comprensible por la JVM, que sirve para indicarle a ésta propiedades vitales acerca del autoejecutable, como por ejemplo dónde buscar la clase principal de la aplicación. El formato de escritura de un archivo de manifiesto puede ser consultado en la referencia [**7**].

El resultado de este proceso será un archivo *jar* que podrá ser ejecutado autónomamente de manera sencilla, en cualquier plataforma que disponga de una Máquina Virtual de *Java*. Bajo entornos *Windows*, por ejemplo, bastará con realizar doble *clic* sobre el archivo *jar*.

Al ejecutarse el archivo *jar*, se extraerán las librerías necesarias y se invocará a la clase principal para la restauración de la agregación. Dado que no habrá CAEAT, y por lo tanto no se dispondrá de tapiz, la interfaz hombre-máquina será aquélla que el usuario definió mediante los componentes gráficos (ver apartado 2.6). Nótese que en este caso la red no será editable, únicamente utilizable, y que los mecanismos para dicha utilización (botones, cuadros de diálogo, etc.) serán los definidos por el creador de la red en el momento de su exportación desde CAEAT.

El archivo *aec* contenido dentro de un *jar* autoejecutable sí será editable si se dispone de la herramienta CAEAT: basta con abrir el autoejecutable desde la interfaz de la herramienta y se podrá trabajar con la agregación como se ha descrito hasta el momento, pudiendo volver a encapsularla a continuación en el archivo autoejecutable.

2.9. EJEMPLO DE CREACIÓN DE UNA AGREGACIÓN

Dada la cantidad de conceptos y funcionalidades que rodean a CAEAT que han sido expuestos, y dado que se hará referencia a ellos en los próximos capítulos, el presente apartado presenta a modo de resumen práctico un ejemplo rápido de creación de una agregación que cubre la práctica totalidad de los mecanismos expuestos en apartados anteriores.

La agregación elegida para ilustrar este ejemplo puede verse en la siguiente figura. Consiste en un Sumador rodeado de componentes que permiten dotarlo de una interfaz gráfica.

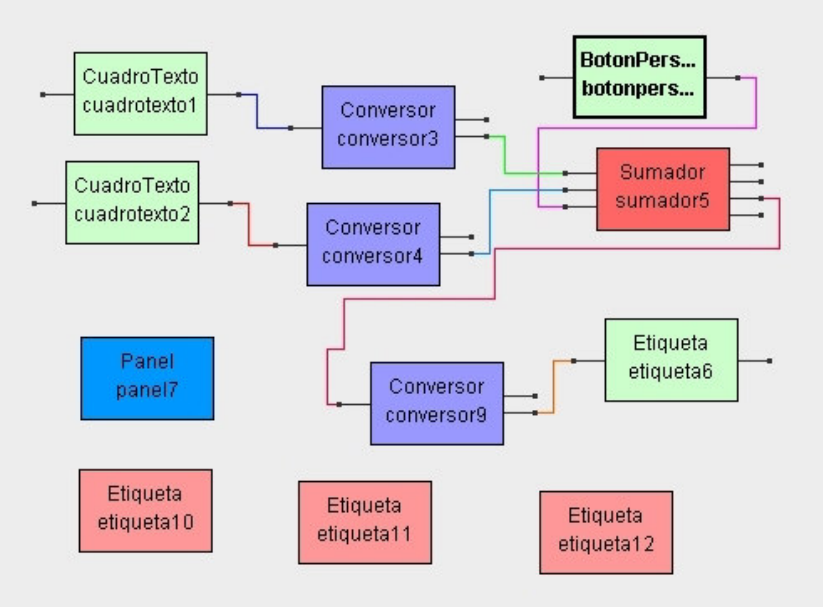

Figura 2.19: Agregación sencilla usada como ejemplo

Sobre el tapiz de CAEAT pueden observarse los siguientes componentes, que han sido arrastrados desde la paleta y convenientemente ubicados e interconectados:

- Dos cuadros de texto (en color verde), que permitirán al usuario introducir un par de números de manera textual.
- Dos conversores conectados a la salida de los cuadros de texto anteriores (en color morado). Se han configurado para proporcionar una salida de tipo entero. Este paso intermedio es necesario puesto que el Sumador necesita entradas de tipo numérico, no textuales (*String*).
- Un botón personalizado (en verde) que se ha rotulado con la leyenda “Sumar” y cuya salida se ha conectado a la entrada “suma” del Sumador, para permitir al usuario dar la orden de sumar a través de él.
- El Sumador propiamente dicho (en color rojo pálido), cuyas entradas son los dos valores a sumar y el *boolean* que da la orden de realizar la suma al ponerse a “verdadero”.
- Una etiqueta a la salida del Sumador (en color verde), para mostrar el resultado de la operación en formato textual no editable. De nuevo, es necesario previamente hacer pasar el valor a través de un conversor que realice la operación inversa a los anteriores: convertir un valor entero a un *String*.
- Elementos gráficos no conectados propiamente a la agregación. Se observan tres etiquetas (en color salmón) que servirán a modo de carteles rotulados con leyendas del tipo “a:”, “b:”, “Resultado:”, para informar al usuario de la naturaleza de cada dato mostrado en la interfaz gráfica. También se incluye el Panel (en color azul), elemento indispensable si se desea construir una interfaz hombre-máquina con la agregación.

En este momento la agregación ya está finalizada. El siguiente paso consiste en la definición de su interfaz de usuario. Se configurará el Panel de manera que contenga todos los elementos con interfaz gráfica que hay sobre el tapiz. El aspecto del Asistente de Creación de Interfaces Gráficas será el que se muestra a continuación:

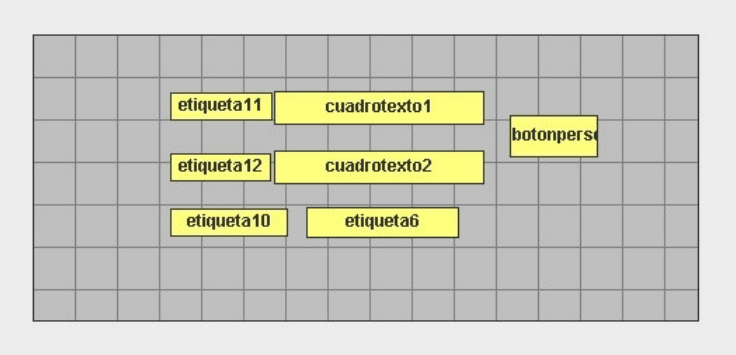

Figura 2.20: Asistente de Creación de Interfaces Gráficas para la red del ejemplo

En la figura anterior, los elementos ya han sido convenientemente arrastrados, posicionados y dimensionados para que la interfaz gráfica resultante sea correcta, intuitiva y elegante. Dicha interfaz gráfica puede verse en la figura siguiente:

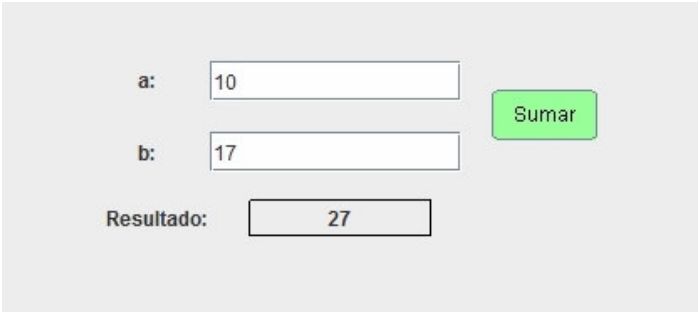

Figura 2.21: Interfaz gráfica resultante tras seguir los pasos del ejemplo

Se debe recordar que en este momento sería posible exportar la agregación a un archivo *jar* autoejecutable. De realizarse este paso, el usuario que ejecuta el archivo *jar* únicamente podrá interaccionar con el *front-end* de la agregación, es decir, la interfaz hombre-máquina que se acaba de montar. Dicho usuario no podrá editar la red ya que (y esto constituye un punto importante de cara al presente proyecto) no llegará jamás a tener conocimiento sobre cómo está implementada.

Parafraseando capítulos anteriores, cabe destacar cómo la agregación final permite realizar una función más ó menos compleja (sumado de dos valores enteros mediante interfaz gráfica) a partir de la colaboración de componentes software muy sencillos: aquí reside la filosofía fundamental de una agregación de componentes software. Las agregaciones de componentes utilizadas en entorno de producción real pueden llegar a ser (y paradigmáticamente suelen serlo) mucho más complejas que la aquí mostrada.

2.10. CASOS DE USO DE LA HERRAMIENTA

Una vez expuestas todas las características con las que contaba la herramienta CAEAT al inicio del presente proyecto, cabe hacer mención de los diferentes roles que programadores y usuarios pueden adoptar alrededor de ella y del concepto más amplio de *SeNetComponents*. La definición de los casos de uso de una herramienta compleja es un ejercicio muy importante que ayuda a definir el modelo de negocio que subyacerá bajo ella una vez lanzada al mercado.

Los tres actores fundamentales que surgen en el entorno que se ha implementado pueden ser representados mediante el siguiente diagrama de casos de uso:

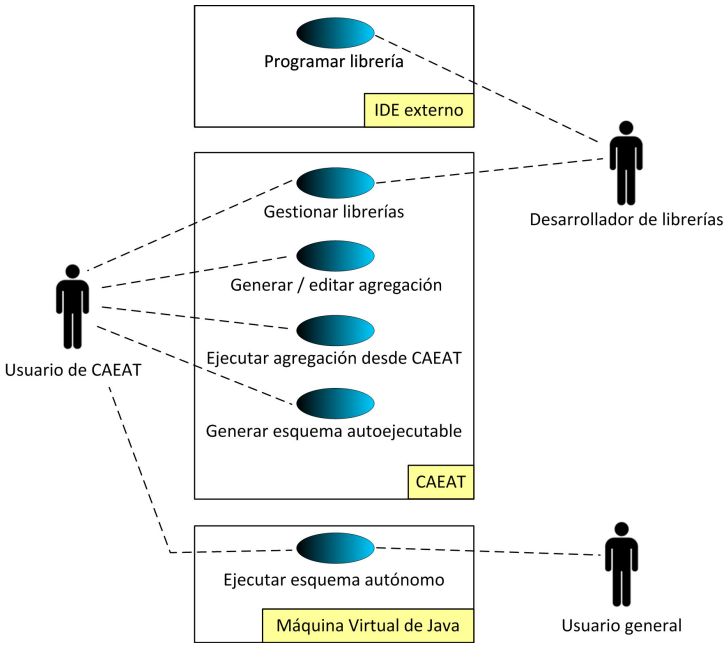

Figura 2.22: Casos de uso del entorno *SeNetComponents*

- **Desarrollador de librerías:** actor encargado de la generación de las librerías de componentes que pueden ser usadas en CAEAT. Llevará a cabo la elaboración de dichas librerías mediante cualquier entorno de edición externo (*Eclipse*, *NetBeans*, etc.). Su modelo de negocio consistirá en la distribución y mantenimiento de los paquetes de librerías desarrollados.
- **Usuario de CAEAT:** el actor principal del entorno diseñado es aquél capaz de crear agregaciones con la herramienta CAEAT. Su primera labor consiste en la adquisición de las librerías de componentes necesarias, con las cuales puede generar agregaciones de componentes. Su modelo de negocio se basa en la edición y creación de agregaciones de componentes, así como de esquemas autoejecutables que distribuirá entre los usuarios generales.
- **Usuario general:** este actor no genera nada, simplemente adquiere las agregaciones de componentes que le son necesarias para realizar una función en su entorno de trabajo, y las ejecuta y despliega mediante la Máquina Virtual de *Java*, sin necesidad de disponer de CAEAT y sin conocimiento de los detalles de implementación de la agregación. Este tipo de actor puede convertirse en un usuario de CAEAT en el momento en que decida adquirir la herramienta para poder editar por sí mismo las agregaciones adquiridas. Cabe destacar que este usuario sin embargo podrá interactuar con las aplicaciones *stand-alone* que ejecute mediante los mecanismos que el creador de las mismas haya definido (interfaz hombre-máquina definida para el autoejecutable en cuestión).

2.11. UTILIDAD PRÁCTICA E INTEGRACIÓN CON APACHE RIVER

En el primer capítulo, cuando se habló de Arquitecturas Orientadas a Servicios, se definió un servicio como una entidad que podía ser usada por un usuario, un programa u otro servicio. En un paradigma SOA, un servicio realiza una operación sencilla bajo demanda, devolviendo unos resultados. Un servicio se debe dar a conocer mediante su interfaz, que define los métodos y tareas que dicho servicio puede desempeñar. Por último, la utilización de un servicio puede realizarse de manera concurrente debido a su falta de estado (pese a que, como se detallará en capítulos posteriores,

habrá ciertas operaciones críticas que se deberán atomizar y proteger contra el acceso concurrente para evitar la corrupción de los datos de proceso del servicio).

La arquitectura explicada en el capítulo anterior se adapta a la perfección a un entorno como el que propone *SeNetComponents*. Únicamente se debe realizar un ejercicio de abstracción y sustituir los componentes generados para trabajar con CAEAT por servicios que habitan por la red y pueden ser utilizados por los usuarios.

2.11.1. Arquitectura distribuida de CAEAT

El objetivo principal del presente proyecto consiste en dotar a la herramienta CAEAT de funcionalidades que la permitan ser una herramienta claramente orientada a servicios. Desde esta nueva óptica, los componentes manejados por CAEAT para construir agregaciones ya no son *beans* encapsulados en un archivo *jar* que hace las veces de librería. Aunque la posibilidad de utilizar únicamente *beans* que residen en una librería en la máquina local debe seguir existiendo, los servicios se solicitan a la red y se pueden utilizar del mismo modo que los *beans* locales.

Para conseguir esta integración se han utilizado las librerías de *Jini / Apache River*, cuyo funcionamiento se ha detallado en el capítulo anterior y cuyos mecanismos de publicación y obtención de servicios se adaptan a la perfección a un entorno como el que propone CAEAT. La herramienta debe ser capaz de desplegar las agregaciones de componentes que se crean con ella como servicios, y de publicar sus interfaces en los servidores de *Lookup* de *Apache River* para darlas a conocer al resto de clientes.

Del mismo modo, CAEAT debe ser capaz de sondear la red y determinar los servicios disponibles en ella, listándolos al usuario como si se tratase de componentes locales. El usuario de CAEAT podrá construir una agregación de componentes mediante la interconexión de servicios que están siendo ofrecidos por la red, con total desconocimiento de su naturaleza o de la entidad física que los está ejecutando. El usuario podrá, si lo desea, interconectar servicios de la red con componentes locales, y la agregación resultante será transparente a este hecho. Incluso será capaz de re-publicar la agregación construida como un nuevo servicio.

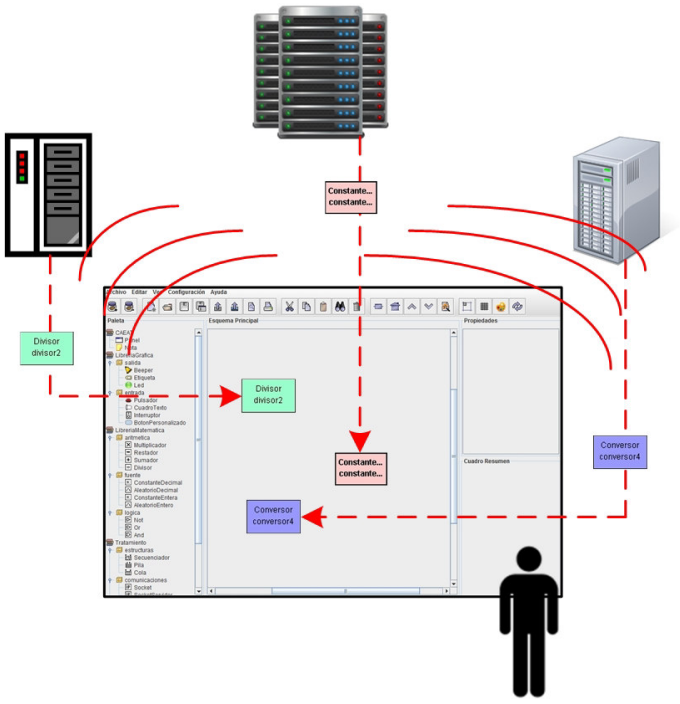

Figura 2.23: Un usuario utiliza CAEAT para sondear la red, obtener y utilizar tres servicios remotos

2.11.2. Aplicación en sistemas SCADA

Hasta este momento, no se ha hecho mención de las posibles aplicaciones prácticas que puede tener la plataforma *SeNetComponents* en general, y la herramienta CAEAT en particular. Las librerías de componentes presentadas son de un propósito lo suficientemente general (operaciones matemáticas, manipulación de datos, etc.) como para no comprometer la utilidad final de la plataforma.

Cabe destacar el hecho de que CAEAT no es una herramienta que sirva para programar visualmente: no permite operaciones de control de flujo, como por ejemplo bucles ó saltos. Simplemente permite generar federaciones de componentes que realizan un cierto procesado en base a la detección de cambios en sus propiedades y que se comunican mediante cadenas de eventos. Este esquema de procesado es muy apropiado para su aplicación en sistemas SCADA.

Un sistema SCADA (*Supervisory Control And Data Adquisition*) permite supervisar y controlar variables de proceso a distancia, proporcionando comunicación con los dispositivos de campo (controladores autónomos) y controlando los procesos de forma automática por medio de un software especializado. Los sistemas SCADA son útiles a la hora de supervisar procesos de producción, realizar controles de calidad, controlar diferentes parámetros físicos de un emplazamiento, etc. Estos sistemas suelen proporcionar interfaces hombre-máquina a través de los cuales lanzan alarmas cuando ha ocurrido un evento de interés. También acostumbran a volcar los datos obtenidos en una base de datos a modo de registro histórico.

Los dispositivos de campo, sensores y actuadores, acostumbran a estar gobernados por elementos hardware como PLC's (*Programmable Logic Controller*) ó PAC's (*Programmable Automation Controller*). Este hardware se encarga de transmitir las medidas realizadas a la unidad maestra a través de la red mediante el protocolo adecuado.

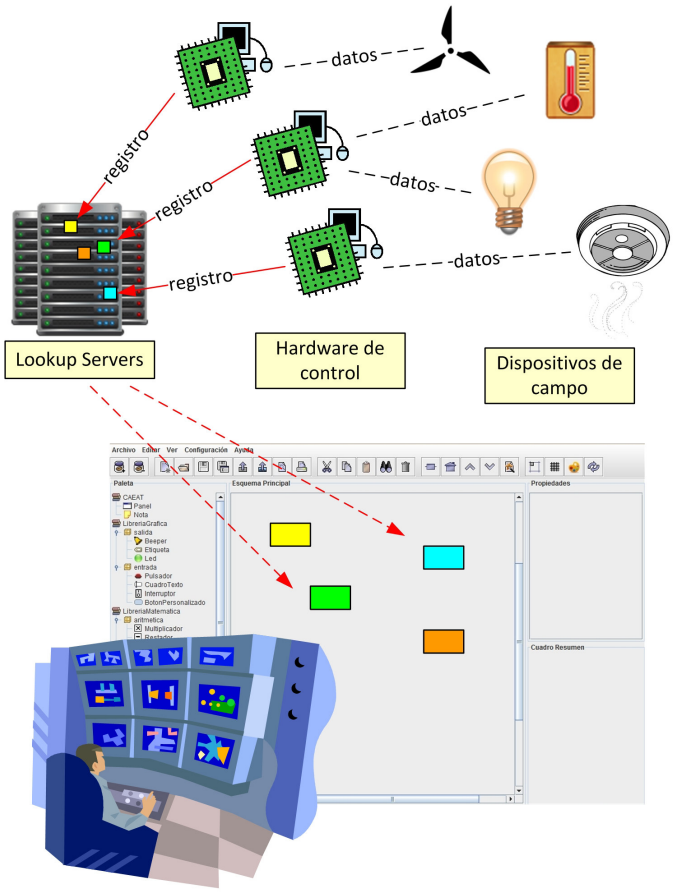

Figura 2.24: Utilización de CAEAT en un entorno SCADA

El rol de CAEAT en un entorno SCADA sería el de la generación de redes completas de supervisión a partir de los servicios simples que representarían a los objetos distribuidos en campo. Para ello, sería necesario que el hardware que monitoriza los dispositivos de campo mapeara las magnitudes medidas en una serie de objetos del lenguaje de programación *Java*. A continuación ese objeto sería ofrecido a la red como un servicio *Jini*, registrando adecuadamente su *proxy* en los servidores de *Lookup* que deberán haber sido desplegados.

Un despliegue de esta naturaleza permitiría generar desde CAEAT redes de supervisión arbitrariamente complejas. Un operario podría visualizar desde CAEAT la arborescencia de servicios disponibles en la red y obtener aquéllos que desea utilizar. Se podrían combinar los servicios obtenidos con elementos de librerías locales que el operario tuviese almacenadas en su máquina. El operario podría generarse una interfaz gráfica para la red de supervisión, como se ha explicado en un apartado anterior, para poder visualizar las salidas de manera agradable. Incluso se podría re-publicar la red acabada de generar como otro servicio, obtenible por otro operario distinto.

Finalmente, un operador de dicho sistema SCADA no tendría por qué conocer que todo su entorno (y el subsistema de adquisición y control en su totalidad) ha sido creado a través de la utilización de *SeNetComponents*, y utilizaría el interfaz gráfico convencional (que se ha diseñado como una agregación de componentes con interfaz hombre-máquina) como si de otro SCADA comercial cualquiera se tratase.

Por la propia definición de servicio *Jini*, los servicios podrían ser utilizados de manera concurrente. Un operario A puede estar utilizando el servicio “Ventilador” con una finalidad, mientras el operario B puede estar usándolo al mismo tiempo para otra completamente diferente. Incluso el objeto *proxy* que permite interactuar con dicho ventilador puede ser insertado en agregaciones completamente independientes. Un cambio en las propiedades del objeto “Ventilador” se reflejará al instante en todas las agregaciones que lo contengan. Del mismo modo, una actuación de un operario sobre el ventilador será notificada de inmediato a todos los operarios que estén haciendo uso de él.

De manera similar, es perfectamente factible la definición de elementos cuyo usuario creador y propietario decida que deben protegerse y de este modo se utilicen de manera exclusiva en una sola agregación de componentes. Hilando un poco más fino, se pueden concebir objetos que pueden ser visualizados universalmente pero sólo pueden modificarse por su legítimo propietario.

Las posibilidades de *SeNetComponents* y CAEAT en el mundo de los sistemas SCADA son muy diversas y están aún sin explorar, puesto que quedan fuera del alcance del presente proyecto.

3. DISEÑO E IMPLEMENTACIÓN DE SERVICIOS REMOTOS

En los anteriores capítulos se han presentado los puntos de partida del presente proyecto. Se ha detallado la naturaleza y usos de la plataforma de agregación de componentes CAEAT, así como los conceptos teóricos que encierra un entorno de programación como *Jini / Apache River*.

En este capítulo se aborda el primer y principal objetivo del proyecto: la adaptación del concepto "servicio", tal y como lo define *Jini / Apache River*, a un entorno de edición visual como CAEAT. Se realiza una descripción de las estrategias de diseño adoptadas para conseguir la publicación y utilización de servicios *Jini* desde la plataforma CAEAT, así como una breve explicación de los detalles de implementación y despliegue de dichos servicios.

3.1. CONCEPTO DE "SERVICIO" EN LA PLATAFORMA CAEAT

El primer paso en la adaptación de las funcionalidades de *Jini* a la plataforma CAEAT consiste en la definición formal y a alto nivel del concepto "servicio" aplicado a dicha plataforma: los requisitos que un servicio debe cumplir, el comportamiento que se espera de él, las operaciones que se podrán llevar a cabo con él, etc. Una vez establecida esta definición será posible bajar un nivel y razonar en términos programáticos para poder decidir la manera en que las clases *Java* que definen un servicio deben ser ampliadas para dar cobertura a los servicios remotos.

En adelante, se hará distinción entre "servicios locales" y "servicios remotos". Los servicios locales son aquéllos que se ejecutan únicamente en la máquina local, tal y como se ha expuesto en el capítulo 2. Los servicios remotos, por el contrario, son aquéllos que se publican y se ponen a disposición de los potenciales clientes de la red, y cuya implementación y manejo constituye la piedra angular del presente proyecto.

En líneas generales, un servicio diseñado para su uso con CAEAT deberá cumplir los siguientes requisitos y características:

1. Implementará una interfaz del lenguaje de programación *Java* que definirá los métodos que el servicio debe ofrecer y que los clientes pueden utilizar.
2. Ofrecerá una clase ó conjunto de clases del lenguaje de programación *Java* que se exportarán como servicio en la máquina proveedora. El concepto de "exportación" fue descrito en el apartado 1.2.4. Esta/s clase/s contendrán la lógica y las propiedades reales del servicio, que los clientes podrán consultar, usar y/o modificar.
3. Durante el proceso de exportación será posible la creación de un objeto *proxy* que encapsulará los detalles de comunicación con el servidor. Este objeto será registrado en los servidores de *Lookup* y puesto por lo tanto a disposición de los potenciales clientes.
4. Incluirá la lógica necesaria para auto-publicarse y auto-gestionarse de una manera suficientemente independiente a la semántica de diseño de CAEAT. La razón es permitir en el futuro la compatibilidad de las librerías desarrolladas con otras plataformas de edición de servicios similares a CAEAT.
5. Un servicio diseñado para CAEAT debe ser comprensible para CAEAT. Según lo descrito en el capítulo 2, esto significa que debe seguir la semántica de *JavaBeans*, respetando todo lo que se ha expuesto: métodos *getter* y *setter*, lanzamiento de notificaciones de cambios de propiedades a los *listeners* (servicios "conectados entre sí"), etc.
6. Respeto a la semántica de *Jini / Apache River* y del estándar *JavaBeans*. Las especificaciones definen con precisión los métodos e interfaces mínimos que un servicio debe implementar para poder ser registrado con éxito como servicio *Jini* en un servidor de *Lookup*. Pese a que se añadan otras características propias de CAEAT en una semántica propia a CAEAT, se deben

respetar las mínimas estipuladas por *Jini*. La razón radica en permitir, en el futuro, que otras herramientas software puedan hacer uso de servicios creados para CAEAT. Si se respeta la semántica *Jini* y *JavaBeans*, una nueva herramienta software que siga dichas semánticas será capaz de hacer uso del servicio, únicamente añadiendo la semántica necesaria para adaptarlo a ella.

3.2. NUEVAS CLASES JAVA IMPLICADAS

Tras el análisis de las características y requisitos anteriores, se llega a la conclusión de que es necesaria la implementación de 5 clases *Java* nuevas para cada servicio que se desee adaptar para uso remoto en el marco de una arquitectura *Jini*. A continuación se realiza una descripción de las funciones desempeñadas por cada clase. Para los casos en que se ofrecen extractos explicativos de código fuente, se ha optado en esta ocasión por un servicio sencillo que calcula el número de letras contenidas en una cadena de caracteres.

El código completo que implementa un servicio sencillo como el mencionado Contador de Letras puede consultarse ampliamente detallado en el apéndice B.

3.2.1. Interfaz remota

Se trata de la interfaz que define los métodos ofrecidos por el servicio. A continuación se muestra un extracto de código de dicha interfaz:

```
public interface ContadorLetrasInterface extends Remote {

        public String getMensaje() throws RemoteException;
        public void setMensaje(String mensaje) throws RemoteException;
        public boolean isContajeAutomatico() throws RemoteException;
        public void setContajeAutomatico(boolean contajeAutomatico) throws RemoteException;
        public boolean isCuenta() throws RemoteException;
        public void realizaCuenta(boolean contar) throws RemoteException;
        public int getLetras() throws RemoteException;
```

Código 3.1: Fragmento de *ContadorLetrasInterface.class*

Como se puede observar, se trata de una interfaz sencilla RMI como la presentada en el apartado 1.2.3.2. Además de los aquí mostrados, la interfaz también obliga a implementar una serie de métodos relativos a la gestión del servicio que son utilizados por CAEAT. Estos métodos se muestran en el apéndice B. Se recomienda su consulta una vez finalizada la lectura de la presente memoria, momento en que se conocerán qué tareas de gestión puede realizar CAEAT sobre los servicios.

Nótese el cumplimiento del requisito (1) del apartado anterior (definición de la interfaz pública del servicio) y el cumplimiento parcial del requisito (6): esta interfaz define métodos que siguen semánticas bien definidas por los estándares (los *getter* y *setter* de *JavaBeans*) además de otros métodos propios de la herramienta CAEAT (los no mostrados aquí). Una hipotética herramienta ajena a CAEAT podría hacer uso de esta interfaz, simplemente usando los métodos canónicos e ignorando los que no siguen una semántica estandarizada (dependientes de la plataforma).

3.2.2. Servidor

Se trata de la clase que se exporta para recibir las peticiones remotas de los clientes. En esta clase, y mediante las llamadas remotas, tiene lugar el procesado real del servicio. Únicamente existirá una instancia de esta clase en toda la red, que estará alojada en la máquina proveedora del servicio. Almacena las propiedades reales del servicio, realiza procesados bajo demanda mediante las llamadas remotas y devuelve los resultados apropiados. También mantiene una lista fresca de los

clientes que en cada momento están haciendo uso del servicio (en el apartado siguiente se detalla el por qué). Esta clase debe implementar los métodos de la interfaz descrita en el apartado anterior.

En el caso del Contador de Letras, el siguiente extracto muestra sus cuatro propiedades y tres de sus métodos: `setMensaje` establece la frase cuyas letras se quieren contar, `realizaCuenta` transmite la petición de cálculo del número de letras, y `getLetras` obtiene el resultado.

```
        //Propiedades del bean:
        private String mensaje;
        private boolean cuenta;
        private boolean contajeAutomatico;
        private int letras;

        public void setMensaje(String mensaje) {
                this.mensaje = mensaje;
                lanzaEvento(1,"mensaje",null);
        }

        public int getLetras() {
                int letras;
                letras = this.letras;
                return letras;
        }

        public void realizaCuenta(boolean cuenta) {
                int nuevoLetras = 0;
                if(cuenta) {
                        StringBuffer buff = new StringBuffer(mensaje);
                        for(int i=0 ; i<buff.length() ; i++) {
                                char c = buff.charAt(i);
                                if( (c >= 65 && c <= 90) || (c >= 97 && c <= 122))
                                        nuevoLetras++;
                        }
                }
                this.cuenta = false;
                this.letras = nuevoLetras;
                lanzaEvento(1,"letras",null);
        }
```

Código 3.2: Fragmento de *ContadorLetrasServer.class*

Recuérdese que en el entorno de CAEAT cada una de las propiedades del servicio será representada visualmente como una patilla. Esto significa que la orden de establecimiento del mensaje puede venir desde un servicio "vecino", la orden de contaje desde otro distinto (por ejemplo, un interruptor) y la salida numérica letras puede servir como entrada a otro servicio distinto.

La función `lanzaEvento` guarda relación con la notificación remota de eventos y será explicada en el apartado siguiente. Cabe destacar que la existencia de esta clase provoca el cumplimiento del requisito (2) de los listados anteriormente (ofrecer una clase o conjunto de clases capaces de exportarse como servicio remoto para llevar en ellas a cabo el procesado real de los datos).

3.2.3. Proxy

El *proxy* es el principal objeto que se registra en los servidores de *Lookup* para poner el servicio a disposición de los clientes. También debe implementar los métodos definidos por la interfaz del servicio.

En el momento de la exportación del *Server*, las API's de *Jini* devuelven un objeto que implementa la interfaz del servicio y que encapsula los detalles de comunicación con el servidor. Este objeto no se debe publicar directamente en los servidores LUS, ya que únicamente actuaría de mero puente entre cliente y servidor, como cualquier otra interfaz RMI. En lugar de eso, *Jini* propone su encapsulación en un objeto *proxy* con una semántica bien definida, que ofrece funcionalidades adicionales como autenticación, mayor seguridad, integridad del código descargado, etc.

Además, la encapsulación de la interfaz remota dentro de un *proxy* permite la invocación de código en local (que se podrá ejecutar previamente a la realización de la llamada remota, o justo después de recibir los resultados), conformando así un objeto conocido como *smart proxy*, que ya fue introducido en el apartado 1.2.3.2. Los servicios desarrollados durante la elaboración del presente proyecto, sin embargo, no han tenido necesidad de utilizar esta característica (pese a que, por diseño, la soportan sin problemas).

```
        final ContadorLetrasInterface inter;

        public String getMensaje() throws RemoteException {
                return inter.getMensaje();
        }

        public void setMensaje(String mensaje) throws RemoteException {
                inter.setMensaje(mensaje);
        }

        public boolean isContajeAutomatico() throws RemoteException {
                return inter.isContajeAutomatico();
        }
```

Código 3.3: Fragmento de *ContadorLetrasProxy.class*

Como se puede observar en el fragmento de código, el cliente que obtenga el *Proxy* realizará las llamadas a los métodos de manera local sobre él, pero éstas acabarán desembocando en llamadas remotas sobre la interfaz que oculta los detalles de comunicación con el servidor (representada por la variable `inter`). La creación de este *proxy* satisface el requisito (3) de la lista anteriormente elaborada (consistente en la obligatoriedad de diseñar un objeto de estas características que se pueda registrar en los servidores de *Lookup*).

3.2.4. Wrapper

Hasta el momento, las clases presentadas han seguido la semántica impuesta por los distintos estándares (*JavaBeans*, *Jini*), por lo que cualquier herramienta diseñada para comprender dichos estándares podría hacer uso de ellos. El *Wrapper* es el primer objeto diseñado exclusivamente para su utilización con una herramienta de edición visual como CAEAT.

La función del *Wrapper* es la de dotar al *Proxy* de una estructura y características idénticas a las de un bean local comprensible por CAEAT y representable en el tapiz, tal y como se mostró en el capítulo anterior. Se debe recordar que el *Proxy* actúa de puente entre clientes y servidor, mientras que un *bean* comprensible por CAEAT debía ofrecer otras características adicionales: lanzamiento de eventos a sus *beans* "vecinos", posibilidad de customización visual (edición de las propiedades) mediante el Customizer, capacidad de introspección (determinación del número y nombre de las propiedades y su naturaleza: sólo lectura, lectura-escritura...) mediante la clase `BeanInfo`, etc.

La clase *Wrapper* ofrece estas características siguiendo una sencilla estrategia: ofrece una estructura idéntica a la de un *bean* local como el mostrado en el capítulo anterior, pero encapsula el *Proxy* hacia el servicio real, y deriva a través de él todas las llamadas a los métodos *getter* y *setter* que hasta el momento se realizaban en el entorno local. Además, se hace cargo de la gestión de las excepciones que las llamadas remotas pudiesen lanzar (una llamada remota, como cualquier función que deba recibir respuesta por parte de la red, está sujeta a fallos). Esta clase, por seguir la semántica de un *JavaBean*, es directamente representable en el tapiz de CAEAT como un componente cualquiera. CAEAT realizará llamadas y peticiones al *Wrapper* como si se tratase de un objeto local, desconociendo que internamente éste las desemboca en un *Proxy* remoto (ya que como se ha mencionado, la gestión de excepciones se realiza de manera interna al *Wrapper*).

La clase *Wrapper* ha sido denominada así precisamente por su función de encapsulación del verdadero *Proxy*, esto es, su labor de "ocultación" de cara a la plataforma que hay por encima y su

gestión interna de los errores y excepciones que pudiesen surgir como consecuencia de la naturaleza cambiante de la red. A continuación se muestran ejemplos de dos típicos métodos *getter* y *setter* adaptados a la nueva estructura del *Wrapper*:

```
        public String getMensaje() {
                String oldMensaje = this.mensaje;
                try {
                        if(!oldMensaje.equals(p.getMensaje())) {
                                this.mensaje = p.getMensaje();
                                cambios.firePropertyChange("mensaje", oldMensaje, mensaje);
                        }
                } catch(RemoteException e) { e.printStackTrace(); }
                return mensaje;
        }

        public void setMensaje(String nuevomensaje) {
                String oldMensaje = this.mensaje;
                try {
                        if(!oldMensaje.equals(nuevomensaje)) {
                                this.mensaje = nuevomensaje;
                                p.setMensaje(nuevomensaje);
                                cambios.firePropertyChange("mensaje", oldMensaje, mensaje);
                                realizaCuenta(contajeAutomatico);
                        }
                } catch(RemoteException e) { e.printStackTrace(); }
        }
```

Código 3.4: Fragmento de *ContadorLetrasWrapper.class*

En este ejemplo, `mensaje` es una propiedad local del *Wrapper* que está en sincronía con la propiedad `mensaje` "real" que se aloja en el servidor (el siguiente apartado detalla la naturaleza de esta sincronía); mientras que `p` representa el *proxy* hacia el servicio. En el caso del *getter*, se consulta y devuelve la propiedad `mensaje` del servidor, aprovechando para actualizar su copia local en caso de que ésta no estuviese ya actualizada. En el caso del *setter*, se envía a través del *proxy* el nuevo mensaje al servidor y se realiza el contaje de letras automáticamente. Nótese que en ambos casos se lanzan eventos de cambio de valor de la propiedad `mensaje`, tal y como se realizaba en el caso de los *beans* locales expuestos en el capítulo anterior.

Resulta evidente que cada cliente del servicio deberá disponer de una copia del objeto *Wrapper*, así como una copia del objeto *Proxy* encapsulada dentro de él. Es por ello que el objeto *Wrapper* también se registra en los servidores de *Lookup* en el momento de la publicación del servicio. El *Proxy* y el *Wrapper* de un mismo servicio quedan asociados en los servidores de Lookup mediante un mecanismo que se detalla en el apartado 3.3.2.3.

Aunque pueda parecer absurdo el registro por separado de dos objetos (*Proxy* y *Wrapper*) que en recepción acabarán siendo encapsulados como uno solo, es muy importante que sea así. Como estipula el requisito (6) de la lista elaborada al inicio del presente diseño, el *Proxy* permite la interacción con el servicio siguiendo la semántica canónica de *Jini* y *JavaBeans*. El *Wrapper*, además, ofrece todas las funcionalidades extra requeridas por una herramienta de diseño visual como CAEAT. Si en el futuro otras herramientas distintas requirieran hacer uso del servicio, el objeto comprensible para ellas sería el *Proxy*. Es posible que estas nuevas herramientas implementen "su propio *Wrapper*" para adaptar la semántica estándar del *Proxy* a su particular entorno de trabajo, tal y como el *Wrapper* de CAEAT lo adapta para poder ser representado como un componente con patillas en el tapiz. Por tanto, el motivo evidente de que exista la necesidad de registrar ambos, es conseguir un desacoplamiento canónico.

3.2.5. Publicador

Tal y como se estableció como objetivo en el requisito (4), un servicio debe incluir la lógica necesaria para auto-gestionarse. Esto no significa que la carga de trabajo necesaria para publicar y mantener un servicio deba recaer sobre el propio servicio: dicha tarea será realizada por la plataforma que publica el servicio (CAEAT en este caso). Sin embargo, con la intención de que en el futuro otras plataformas puedan publicar los servicios inicialmente diseñados para CAEAT de manera sencilla, se ha creado la clase *Publicador*.

Esta clase ayuda a hacer abstracción entre el servicio (sus objetos *proxy*, *wrapper*, etc.) y la lógica de funcionamiento de la plataforma que lo está publicando. Para ello, implementa una interfaz del lenguaje de programación *Java*, llamada `PublicadorInterface`, que define claramente los métodos que se deben invocar para gestionar un servicio: publicarlo, detenerlo, cambiar su fecha de caducidad...

El contenido completo de esta clase se muestra en el anexo B, y es recomendable consultarlo una vez conocidas las operaciones de gestión que es posible realizar sobre un servicio, que se presentan en el capítulo 6. Por el momento, se muestra a modo de ejemplo la función que se encarga de la exportación y publicación de un *bean*:

```
public Object[] publicarBean(Object beanServer, Object beanWrapper,
        ServiceRegistrar[] servers, String[] groups, ActionListener CSMListener) {

        if (servers.length == 0) {
                return null;
        }
        lookupServers = new ArrayList<ServiceRegistrar>(Arrays.asList(servers));
        gruposRegistro = groups;
        this.beanWrapper = beanWrapper;
        bje = getExporter();
        ((ContadorLetrasServer)beanServer).setCSMListener(CSMListener);
        try {
        inter = (ContadorLetrasInterface)bje.export((ContadorLetrasServer)beanServer);
        } catch (ExportException e) { e.printStackTrace(); }

        proxy = ContadorLetrasProxy.crear(inter);
        try {
                RegistraLUS(proxy,new Entry[]{tipoServicio},servers,false);
                ServiceInfo siWrap = new
                ServiceInfo(null,null,null,null,null,serviceIDProxy.toString());
                ServiceInfo[] siWrapArray = new ServiceInfo[]{siWrap};
                ((WrapperInterface)beanWrapper).setServiceID(serviceIDProxy);
                RegistraLUS(beanWrapper,siWrapArray,servers,false);
        } catch(IOException e) { e.printStackTrace(); }
        isPublicado = true;
        return new Object[]{proxy, serviceIDProxy, this};
}
```

Código 3.5: Fragmento de *ContadorLetrasPubli.class*

Pese a que este ejemplo incluye operaciones y parámetros que aún no se han explicado (se hará en los apartados siguientes), su función es la de presentar la filosofía de funcionamiento que se ha pretendido implementar para la clase *Publicador*. El método `publicarBean` admite una serie de objetos como parámetros: la clase *Server*, la clase *Wrapper*, las referencias a los servidores LUS en los que se debe registrar el servicio, etc. Este método realiza internamente los procedimientos necesarios para publicar el servicio, y devuelve una serie de objetos en consecuencia.

Queda a criterio de la plataforma publicadora del servicio (CAEAT en este caso) la forma en que los servidores de *Lookup* son buscados, el momento y la forma en que este método debe ser llamado, las actuaciones a realizar sobre los objetos devueltos, etc. La mayor carga de trabajo en la gestión de un servicio, pues, recae sobre la plataforma: el servicio únicamente proporciona la lógica para su gestión, y es responsabilidad de la plataforma gestora su correcto uso. Dado que la interfaz `PublicadorInterface` define claramente las operaciones de control de flujo que se pueden realizar sobre los servicios, los parámetros que se le deben proporcionar, y los resultados devueltos

por cada operación, se consigue desacoplar de una forma muy eficiente la capa del servicio de la capa de la plataforma CAEAT, permitiendo en el futuro el desarrollo de otras plataformas similares que hagan uso de los mismos servicios aquí desarrollados.

3.2.6. Diagrama de clases

Se presenta a continuación un diagrama de clases que ilustra la interrelación entre las nuevas clases implementadas y su relación y correspondencia con las tres clases que se presentaron en el capítulo anterior y que constituían los *beans* locales. Las clases nuevas son representadas en color azul, mientras que las ya existentes se muestran en color amarillo:

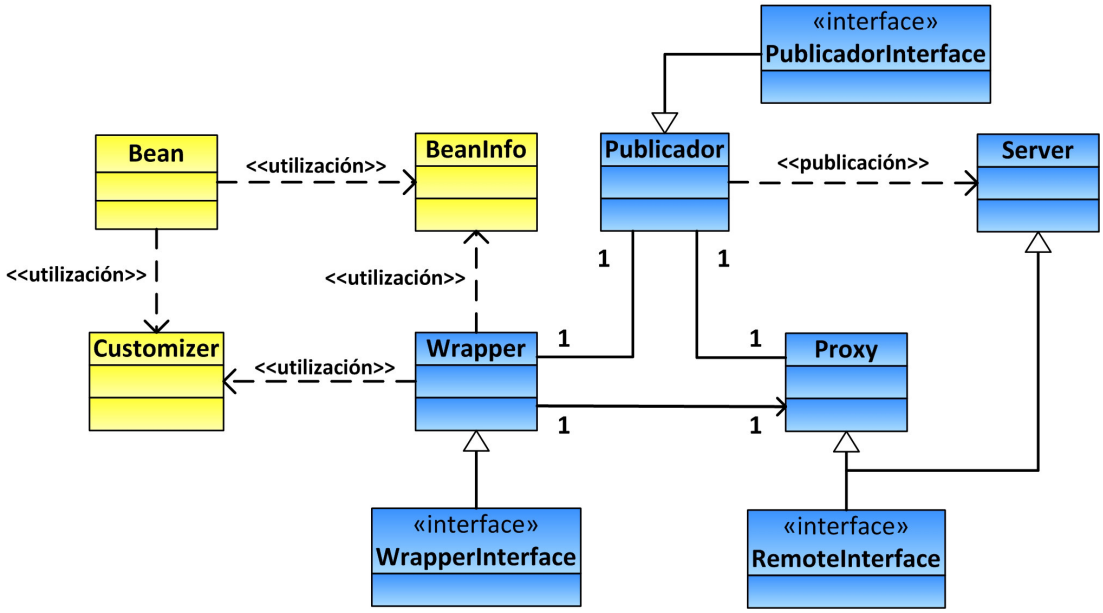

Figura 3.1: Diagrama de clases de un servicio remoto

Se ha denotado en el diagrama anterior como *RemoteInterface* la interfaz remota del servicio presentada al inicio de este apartado, que será distinta para cada uno de los servicios. Como se puede observar, es posible utilizar las mismas clases *BeanInfo* y *Customizer* para la versión remota del servicio. Dado que los métodos y propiedades ofrecidos por la clase *Wrapper* serán los mismos que los que ofrecía el *bean* local (salvo que se tratará de la versión remota de éstos), la utilización de ambas clases sigue siendo válida siempre que se lleven a cabo mínimas modificaciones en el código para determinar si se está utilizando la clase local (*Bean*) o la remota (*Wrapper*).

3.3. ARQUITECTURA DE PUBLICACIÓN DE UN SERVICIO EN CAEAT

Una vez descritas las cinco nuevas clases implicadas en la adaptación a un entorno remoto de los *beans* diseñados para CAEAT, este apartado aborda su utilización en el marco de la plataforma CAEAT para poder publicar y desplegar servicios *Jini*. Se detalla asimismo la utilización de las herramientas y procedimientos del entorno de programación *Jini / Apache River* que permiten la publicación y el mantenimiento de dichos servicio.

3.3.1. Movilidad y ubicación de los objetos

En el capítulo 1.2.4 se expuso de manera genérica el proceso de exportación y publicación de un servicio en el marco del entorno de programación *Jini / Apache River*. Dado que se han presentado todas las clases *Java* que conforman un servicio remoto de CAEAT, se concreta aquí su utilización y la

movilidad de los objetos que tiene lugar durante los procesos de publicación y obtención de un servicio. Se usa el mismo escenario simplificado que en el ejemplo del mencionado capítulo: un único proveedor, un único servidor LUS y un único cliente que obtiene el servicio.

1. **Exportación del servicio y creación del *Proxy* y del *Wrapper***: tras una búsqueda de servidores LUS por la red, CAEAT procede a la exportación del servicio. La clase *Server*, aquélla que contiene las variables de proceso "reales" del servicio, se marca como accesible por los clientes (se asocia a un puerto libre de la máquina proveedora). Durante el proceso, las API's de *Jini* crearán un objeto que encapsulará los detalles de comunicación con el servidor. Un objeto *Proxy* es creado mediante el uso de ese objeto. También se prepara un nuevo objeto *Wrapper*.

2. **Registro de *Proxy* y *Wrapper* en los servidores LUS**: los objetos recién creados son registrados secuencialmente en los servidores de *Lookup* que se desee. Por cada objeto registrado, el servidor LUS devuelve a la entidad publicadora un `ServiceID` (identificador universal de servicio) y un `Lease` (concesión temporal), objetos útiles para la gestión y mantenimiento del servicio cuya naturaleza es explicada con más detalle en el siguiente apartado. Cabe destacar la posibilidad de registrar servicios junto a los denominados atributos de *Entry*: objetos que encapsulan información en formato agradable para el ser humano (nombres, números, descripciones...) y que facilitarán las búsquedas a los clientes.

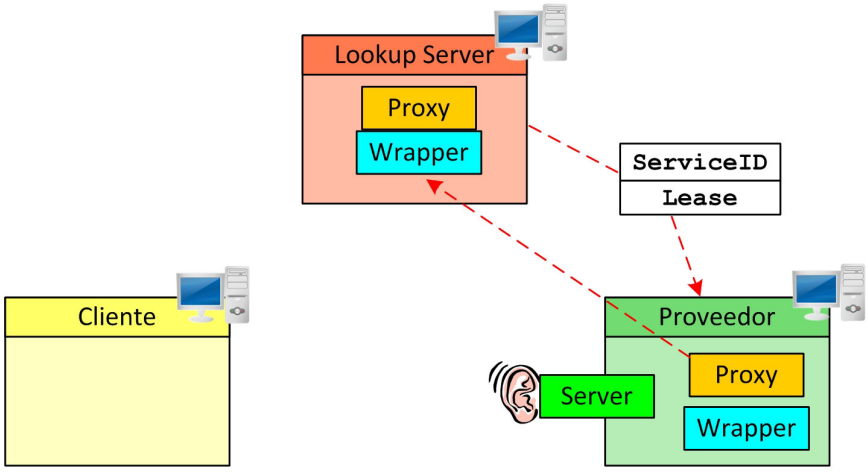

Figura 3.2: Proceso de exportación y publicación de un servicio

3. **Transcurso del tiempo**: a partir de este momento, el servicio será visible en las búsquedas que los clientes realicen a los servidores LUS. El proveedor del servicio es responsable de su mantenimiento. Puede detenerlo en el momento que lo desee mediante la cancelación de su *Lease* o concesión. Puede asimismo pedir la renovación de ésta mediante una solicitud al servidor LUS a través del propio objeto `Lease`.

4. **Obtención por parte de un cliente**: los clientes realizan la búsqueda de servicios mediante su *Proxy*. Es posible realizar búsquedas usando "plantillas", esto es, definiendo clases e interfaces que los servicios encontrados deberán heredar / implementar, o mediante la elección de los atributos de *Entry* que el servidor pueda haber establecido. En el caso del presente proyecto los *proxies* implementados heredan de la clase `SimpleBeanInfo`. Además, un atributo de *Entry* establecido bajo el nombre "CAEAT_Service" indica que ese *Proxy* es apto para usarse con la plataforma CAEAT. En este último caso se debe buscar también el *Wrapper*. El *Wrapper* se habrá registrado con el parámetro `ServiceID` del *Proxy* como atributo de *Entry* (más información en el siguiente aparatado), por lo que habrán quedado unívocamente asociados y su localización será inmediata. Con las copias de los dos objetos en su poder, CAEAT es capaz de encapsular el *Proxy* dentro del *Wrapper*, realizar una

actualización inicial de los atributos del servicio y dejar el objeto resultante listo para ser insertado en el tapiz como un componente más.

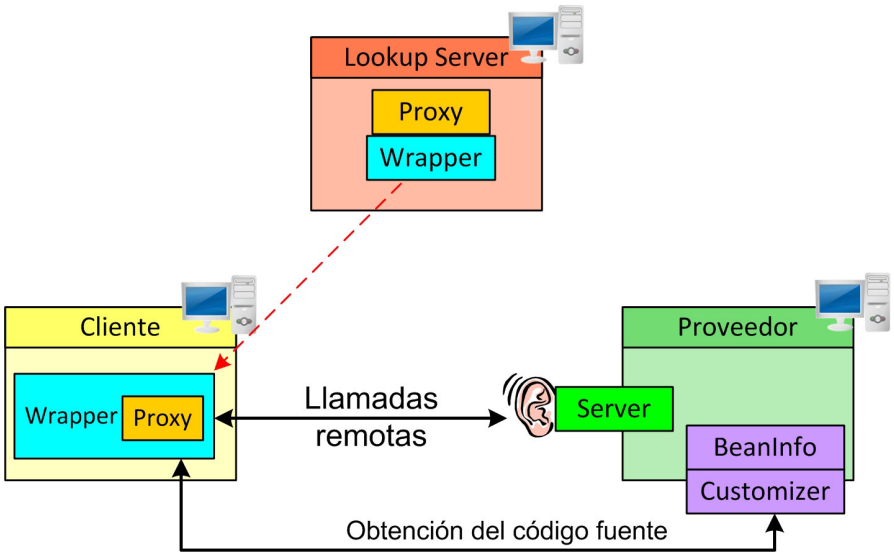

Figura 3.3: Proceso de obtención del servicio por parte de un cliente

5. **Búsqueda y carga del *BeanInfo* y el *Customizer***: al igual que los servicios locales expuestos en el capítulo 1, los servicios remotos requieren de las clases `BeanInfo` y `Customizer` para poder ser analizados y editados visualmente. Sin embargo, el cliente no necesita disponer de estos objetos, simplemente ha de conocer su código fuente (el archivo de extensión *class* debe ser accesible por la Máquina Virtual de *Java*). Para ello, el estándar *JavaBeans* proporciona la clase `Introspector`, que determina dónde se deben buscar archivos de este tipo. En el momento de la obtención de un servicio, se añade a dicha clase una dirección que apunta a una localización en la que el servidor habrá copiado el código fuente necesario. Esta dirección se obtiene mediante consulta al *codebase* de los objetos obtenidos, un concepto que se explica a continuación.

3.3.2. Requisitos de despliegue de un servicio

Este apartado discute algunos de los conceptos técnicos, tanto del lenguaje de programación *Java* como del entorno *Jini / Apache River*, que se deben tener en cuenta a la hora de desplegar un servicio remoto, y que en algunos casos ya han sido brevemente mencionados en apartados anteriores.

3.3.2.1. Codebase

El *codebase* es una propiedad de la Máquina Virtual de *Java* que debe establecer adecuadamente todo programa que pretenda ofrecer la posibilidad de recepción de peticiones remotas por parte de clientes. Tal y como se avanzó en el apartado 1.2.3, cuando un objeto del lenguaje de programación *Java* es transmitido por la red, únicamente se transmite su estado actual: el valor de sus propiedades en el momento de la transmisión. Se da por hecho que la JVM receptora conoce el código fuente de la clase cuya instancia es el objeto transmitido.

La Máquina Virtual de *Java* no puede ejecutar código proveniente de una clase que no haya sido resuelta. “Resolver” una clase implica tomar el archivo *class* que conforma la clase, analizarlo y cargarlo en memoria para su utilización. Si la JVM receptora no cuenta en su entorno local con una copia del código fuente (del archivo *class*) de la clase en cuestión, no podrá utilizar un objeto descargado dinámicamente en tiempo de ejecución.

En un esquema funcional con gran movilidad de código como el que implementa este proyecto, las máquinas receptoras (clientes) desconocen por completo la naturaleza de las clases cuyos objetos intentan utilizar remotamente, por lo que las máquinas proveedoras del servicio deben ofrecer un mecanismo para compartir la información de los archivos *class* de dichas clases.

Todo objeto convertido a un flujo de *bytes* y transmitido por la red es "anotado" con la propiedad `java.rmi.server.codebase`, si ésta ha sido convenientemente establecida. Esta propiedad contiene una URL que apunta a una localización en la red en la cual presumiblemente el cliente puede obtener la definición de la clase que le falta. Generalmente, la máquina proveedora del servicio habrá establecido la propiedad *codebase* de manera que apunte a una localización en la que se almacenarán los archivos *class* y demás recursos que deban ser compartidos con los clientes. Dichos recursos serán compartidos a través de la red mediante un servidor HTTP o FTP (ver apartado siguiente).

```
int puerto = 8080;
try {
        String IP = java.net.InetAddress.getLocalHost().getHostAddress();
        String codebase = "http://" + IP + ":" + puerto + "/";
        System.setProperty("java.rmi.server.codebase", codebase);
} catch(UnknownHostException e) {
        e.printStackTrace();
}
```

Código 3.6: Establecimiento de la propiedad *java.rmi.server.codebase* en CAEAT

Cabe remarcar que *codebase* no es un parámetro de ninguna función del lenguaje *Java*, sino una propiedad intrínseca de la *Java Virtual Machine*, que es válida durante todo el tiempo de vida de la misma. Estas propiedades pueden ser consultadas y modificadas siempre que la entidad que lo realice tenga permiso para hacerlo (no se puede, por ejemplo, cambiar la propiedades de una JVM remota).

El *Classpath* de una aplicación *Java* se define como el conjunto de clases a las que dicha aplicación tiene acceso, es decir, aquéllas que puede resolver, cargar en memoria, instanciar y utilizar. Un uso correcto de la propiedad *codebase* puede ser vista como una ampliación en tiempo de ejecución del *Classpath* de una aplicación, una herramienta extremadamente potente en entornos distribuidos como *Jini*.

La descarga y posterior intento de utilización de un objeto cuya definición de clase no puede ser obtenida (por ejemplo, porque la propiedad *codebase* no ha sido correctamente establecida, o porque no hay ningún servidor en el lado opuesto ofreciendo los archivos *class*) desemboca en el lanzamiento de una `ClassNotFoundException`, y desencadena que el objeto no se pueda utilizar.

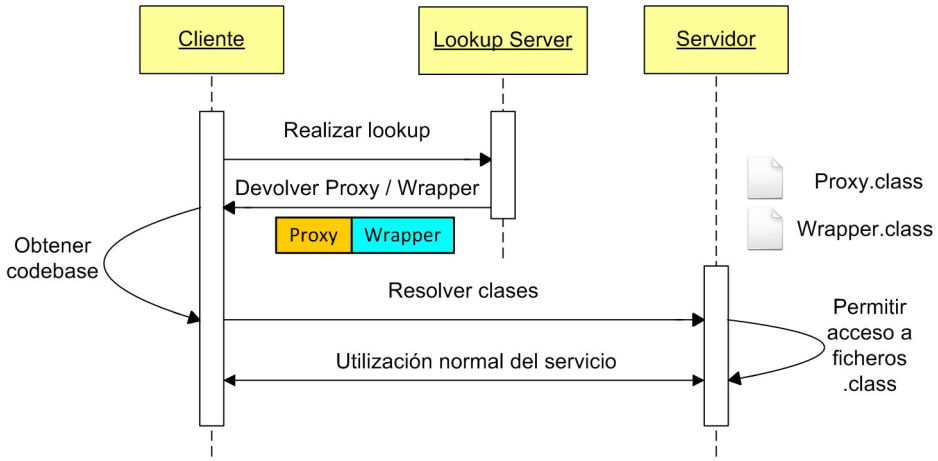

Figura 3.4: Proceso de obtención de objetos remotos y resolución de sus clases

3.3.2.2. Servidor HTTP

Como ya se ha comentado, toda aplicación que pretenda implementar la descarga dinámica de código debe ofrecer acceso a los archivos *class* de las clases implicadas. Esto se puede realizar mediante un servidor HTTP o un servidor FTP.

La distribución de *Apache River* incluye una sencilla implementación de un servidor HTTP, ligero, escrito en el lenguaje de programación *Java* y fácilmente ejecutable y configurable. El servidor no dispone de interfaz gráfica para que sea fácilmente encapsulable como complemento dentro de otro proyecto. Para CAEAT se ha optado por la utilización de dicho servidor, pero no se ha encapsulado dentro de la plataforma para no restringir opciones futuras (puesto que se quiere dotar al usuario que despliegue CAEAT de la libertad de usar el servidor HTTP que crea más oportuno).

La estrategia usada por CAEAT para compartir los ficheros *class* consiste en copiar todos los recursos que se deban compartir a una carpeta llamada "CAEAT-Public" que a su vez es configurada como la carpeta principal del servidor HTTP. Es importante configurar el servidor HTTP para que trabaje en el mismo puerto con el que se ha configurado la propiedad *codebase* (en el ejemplo del apartado anterior, el puerto 8080). Se debe recordar que los clientes que obtengan el servicio intentarán obtener remotamente la definición de sus clases en la localización apuntada por dicha propiedad. Si en su establecimiento se ha especificado un puerto (por ejemplo, el mencionado 8080), se debe garantizar que el servidor HTTP se encuentra efectivamente escuchando las peticiones entrantes en ese puerto. En el entorno de CAEAT, los recursos a compartir incluyen las clases `BeanInfo` y `Customizer`: como se ha mencionado en el capítulo anterior, éstas deben estar disponibles en el *Classpath* de la máquina cliente que hace uso del servicio.

Por último, cabe remarcar que la máquina que aloja el / los servidores de *Lookup* también debe ejecutar un servidor HTTP, pese a no actuar como proveedora de ningún otro servicio. La implementación de un servidor de *Lookup* no es otra cosa que otro programa *Jini* dentro de la federación, y realiza muchas de las funciones descritas hasta el momento tal y como se ha explicado: exportación de código para atender llamadas remotas de los clientes, movilidad de objetos, etc. Se debe tener en cuenta que el servidor LUS transmite al resto de entidades objetos que son instancias de clases que los clientes podrían perfectamente desconocer.

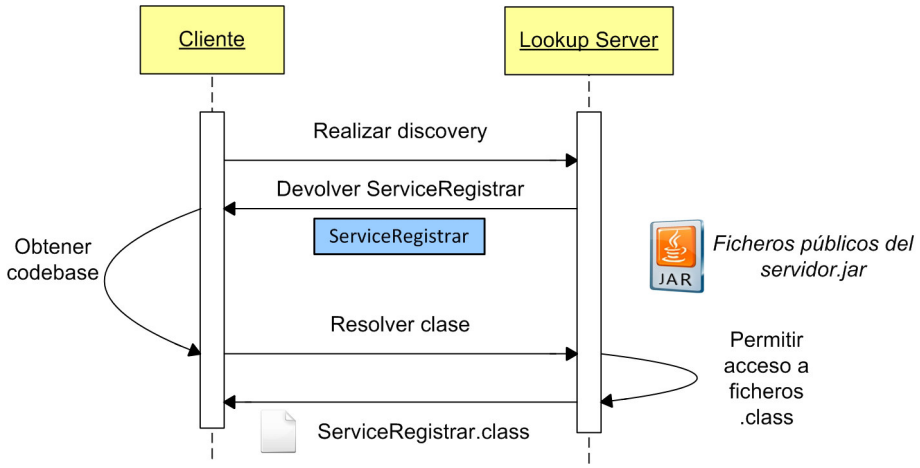

Figura 3.5: Resolución de las clases que implementan un *Lookup Server*

En el ejemplo ilustrado por la figura anterior, un cliente realiza un proceso de *discovery* y el servidor LUS le remitu un objeto de la clase `ServiceRegistrar` que le sirve para comunicarse con él. No se puede asumir que el cliente posea y conozca todas las clases del *framework Jini / Apache River*, por lo que se resuelve ésta remotamente tal y como se ha detallado en el apartado anterior. Es por ello que una parte del código del servidor LUS también debe ser accesible desde el exterior. El servidor HTTP ofrecido por *Apache River* ya contempla en su configuración por defecto el servicio de las clases del servidor de Lookup encapsuladas en un archivo *jar*.

3.3.2.3. ServiceID

El parámetro `ServiceID` encapsula un valor de 128 bits que representa unívocamente y de manera universal un servicio *Jini* dentro de todas las federaciones de servicios *Jini* existentes. Este parámetro es asignado automáticamente por el servidor de *Lookup* a un servicio en el momento de su registro, y devuelto a la entidad que ha realizado el registro. También es posible registrar un servicio usando un `ServiceID` previamente guardado (por ejemplo, cuando se trata de un re-registro de un servicio que ya había tenido un `ServiceID` asignado durante su primer registro).

Puede llamar la atención el hecho de que el parámetro pretenda ser universal para todos los servicios de todas las federaciones *Jini* habidas y por haber. Para conseguir esto, la generación del `ServiceID` está sustentada en dos pilares básicos:

- Una secuencia de bits aleatoria cuya semilla se genera a partir del propio objeto que se está registrando en el *Lookup Server*. Esto garantiza que el registro de servicios que difieran en algo, por mínimo que sea, reciban diferentes identificadores.
- Una segunda secuencia de bits dependiente de una marca temporal con resolución de 0,1 milisegundos. Esto garantiza que el registro consecutivo de servicios idénticos generen identificadores diferentes, a menos que los registros se realicen dentro de un intervalo de 100 nanosegundos, algo imposible. La marca temporal es suficientemente larga como para garantizar la universalidad de los `ServiceID`'s hasta el año 3.400 aproximadamente.

Se debe puntualizar que en ningún caso se permite el registro de servicios mediante el establecimiento manual del idénticos `ServiceID`'s. Si el servidor de *Lookup* recibe una petición de registro de un servicio con un `ServiceID` (establecido por el usuario) que coincide con otro `ServiceID` ya asignado a otro servicio, esto se interpreta como una actualización del servicio: la nueva versión será reemplazada por la que ya se encontraba en el servidor LUS.

En el entorno CAEAT, se registran dos objetos en los servidores de *Lookup* por cada servicio desplegado: el *Proxy* y el *Wrapper*. Aunque se reciben, por lo tanto, dos `ServiceID`'s distintos, únicamente se almacena y mantiene el del *Proxy*. A la hora de registrar el *Wrapper*, el `ServiceID` del *Proxy* se registra como atributo de *Entry* del *Wrapper*. De esta manera, *Proxy* y *Wrapper* quedan universalmente asociados, y en el momento de la búsqueda, al obtener el *Proxy*, la obtención del *Wrapper* es inmediata si se solicita a los servidores LUS un objeto que tenga como atributo de *Entry* el `ServiceID` del *Proxy* recién obtenido.

3.3.2.4. Lease

El objeto `Lease` representa la concesión temporal que un servidor de *Lookup* realiza ante un registro. En otras palabras, representa el tiempo durante el cual el servidor LUS mantendrá el objeto en su mapa interno de objetos registrados: transcurrido ese tiempo, el registro caducará y el objeto será descartado. Los objetos `Lease` son devueltos a las entidades que publican servicios en el momento del registro de los objetos en los servidores de *Lookup*.

Lejos de ser una simple marca temporal, el objeto `Lease` encapsula funcionalidades extremadamente útiles para el mantenimiento de los servicios. A través de él, es posible solicitar explícitamente la cancelación abrupta de la concesión, para poder des-publicar un servicio a voluntad. También es posible consultar el tiempo de registro restante del servicio, y solicitar una extensión el mismo (alargar la fecha de caducidad).

Nótese que para realizar tales solicitudes la entidad publicadora y gestora debe de comunicarse con el servidor LUS. En efecto, el objeto `Lease` es, en realidad, un *proxy* remoto hacia el servidor de *Lookup*. Igual que cualquiera de los *proxies* de la plataforma *Jini* expuestos hasta el momento, "esconde" los detalles de comunicación con dicho servidor: ubicación, dirección, nombre, etc.

3.3.2.5. Servidor de Lookup Reggie

La distribución de *Apache River* ofrece diferentes implementaciones de distintos servidores de *Lookup*. "Reggie" es el nombre que recibe la implementación que se ha utilizado durante la elaboración del presente proyecto. *Reggie* es una de las implementaciones de servidores LUS más sencillas a la vez que más altamente configurables a través de sus archivos de configuración. Como en el caso del servidor HTTP, *Reggie* se ofrece sin interfaz gráfica para facilitar su ensamblaje, si se desea, dentro de otro proyecto software (y como en el caso del servidor HTTP, no se ha llevado a cabo esta acción en CAEAT por los mismos motivos que se adujeron).

Como todo servidor de *Lookup* del entorno *Jini / Apache River*, *Reggie* soporta su afiliación a grupos de *Lookup*. En el capítulo 1 se explicó que un grupo de *Lookup* era un nombre al cual ese servidor quedaba asociado. Las búsquedas de servicios, realizadas mediante mensajes *multicast*, se realizan sobre un conjunto de grupos (o sobre todos los grupos disponibles). Únicamente los servidores LUS afiliados a alguno de los grupos que se solicitan en la búsqueda responderán a dicha solicitud. La gran ventaja de *Reggie* radica en que soporta su afiliación a más de un grupo de búsqueda simultáneamente, proporcionando así una gran libertad de alternativas de despliegue.

La figura siguiente ejemplifica el comportamiento explicado mediante la representación de cuatro servidores de *Lookup* distintos. Tres de ellos están asociados a los grupos de *Lookup* "A", "B" y "C" respectivamente, mientras que el cuarto pertenece simultáneamente a los grupos "A" y "C". Dos clientes de la federación de servicios realizan búsquedas sobre distintos grupos de *Lookup*, respondiendo únicamente los servidores que pertenecen a alguno de los grupos buscados. Se debe recordar que siempre existe la posibilidad de realizar las búsquedas sobre todos los grupos, en cuyo caso responderá la totalidad de servidores de *Lookup* que reciban la solicitud, sin atender a su afiliación.

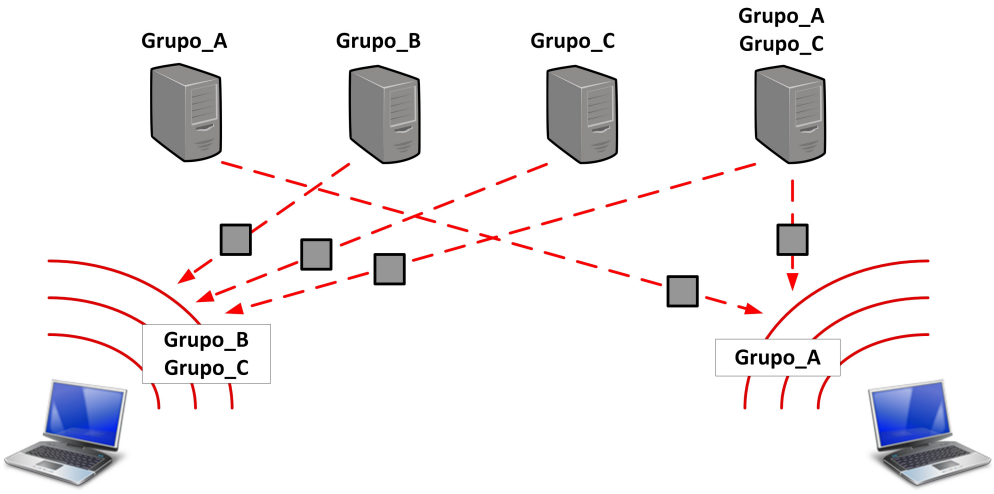

Figura 3.6: Ejemplo de despliegue y búsqueda de servidores de Lookup según su grupo de registro

Se debe mencionar el particular comportamiento de *Reggie* a la hora de conceder un *Lease*, que afecta sobremanera al tratamiento que se debe hacer de estos objetos. En el momento del registro de un objeto, la entidad publicadora solicita a *Reggie* un tiempo de concesión deseado. *Reggie* nunca concederá el tiempo solicitado, sino uno menor que puede ser configurado (dentro de unos límites) por el usuario que despliega *Reggie* en el momento de su lanzamiento. Es responsabilidad de la entidad publicadora del servicio velar por la periódica renovación de ese *Lease*, en caso de que se desee que el servicio siga publicado en el servidor LUS.

En caso de que *Reggie* contenga una gran cantidad de objetos registrados en él, se empezarán a conceder *Leases* de duración más larga. Esto se realiza para evitar sobrecargar la red con peticiones continuas de renovación de *Lease* por parte de las entidades publicadoras de servicios, en caso de que haya una gran densidad de éstas, y de servicios, viviendo en la federación *Jini*.

Las especificaciones completas de *Reggie* y sus distintos parámetros de configuración pueden consultarse en la referencia [**8**].

3.3.2.6. SecurityManager y SecurityPolicy

Por defecto, y por razones de seguridad, la Máquina Virtual de *Java* no permite la ejecución de código descargado de manera dinámica desde localizaciones remotas, de la misma manera que no permite a una aplicación llevar a cabo acciones potencialmente peligrosas o inseguras para el sistema. Para permitir la ejecución de dichas tareas potencialmente peligrosas, es necesaria la instalación de un gestor de seguridad y la definición de una política de seguridad.

- ***SecurityManager***: se trata de una clase que permite a la aplicación, previamente a la ejecución de una acción potencialmente peligrosa, determinar si esta acción se está llevando a cabo en un contexto en el cual está permitida. Si no lo está, el *SecurityManager* no permite a la aplicación ejecutar dicha acción. Para determinar qué acciones están permitidas y cuáles no, el *SecurityManager* debe ir acompañado de una política de seguridad (se explica a continuación).

  La instalación de un gestor de seguridad es obligada en una plataforma como CAEAT, y se lleva a cabo mediante llamada a una función de sistema:

  ```
  System.setSecurityManager(new java.rmi.RMISecurityManager());
  ```

- ***SecurityPolicy***: se trata de un fichero que es consultado por el *SecurityManager* para determinar qué acciones es posible llevar a cabo y cuáles no. Por defecto, todas las acciones potencialmente peligrosas se encuentran prohibidas, y se deben añadir cláusulas al fichero *SecurityPolicy* para denotar explícitamente aquéllas que sí pueden ser llevadas a cabo. La instalación de una política de seguridad se realiza mediante el establecimiento de una propiedad de la Máquina Virtual de *Java*:

  ```
  System.setProperty("java.security.policy",<ruta_del_fichero>);
  ```

  Las cláusulas que es posible incluir en el fichero de política de seguridad abarcan una gran cantidad de acciones tales como la aceptación de conexiones entrantes, el permiso de escritura en determinadas localizaciones, el lanzamiento de nuevos *Threads*, etc.

  El presente proyecto no se ha marcado como objetivo el análisis de todos los posibles permisos que puede llegar a necesitar un usuario de la plataforma CAEAT, por lo cual se ha usado la política de permisos mostrada a continuación:

  ```
  grant{
        permission java.security.AllPermission;
  };
  ```

  Código 3.7: Política de permisos laxa aplicada en la plataforma CAEAT

  Esta política de permisos es la más laxa que existe, ya que permite llevar a cabo cualquier acción. Aunque es muy útil en tiempo de desarrollo, no sería factible su uso una vez la herramienta esté funcionando en un entorno de producción, pues resultaría potencialmente peligrosa.

3.4. SINCRONIZACIÓN DE VARIABLES Y EVENTOS REMOTOS

De la arquitectura diseñada en el apartado anterior se desprende que las propiedades “reales” del servicio son únicas a lo largo de la red, y se encuentran encapsuladas en la clase *Server* que se exporta en la máquina proveedora del servicio. Los clientes obtienen una copia del *Proxy* y el *Wrapper* del servicio, y por lo tanto disponen en su espacio local (en su tapiz de CAEAT) de una copia de las variables del servidor.

Es evidente que se hace necesario un mecanismo de sincronización, puesto que en un entorno de utilización concurrente de servicios como *SeNetComponents* es uno de los objetivos primordiales que todos los clientes trabajen sobre el mismo servicio final, y que la actuación de un cliente sobre dicho servicio sea reflejada al instante en la visión que el resto de clientes tienen del servicio.

Para conseguir tal fin, el entorno de programación *Jini / Apache River* ofrece API’s relacionadas con el lanzamiento y recepción de eventos remotos.

3.4.1. Descripción de la problemática

De acuerdo a la arquitectura descrita hasta el momento, un escenario típico de aplicación sería el que se ejemplifica a continuación:

1. Un servidor ha exportado y publicado un servicio para su utilización. Este servicio contiene un atributo de tipo entero, llamado “a”, de valor igual a 5.
2. Diferentes clientes han buscado el servicio en los servidores de *Lookup*, han obtenido los objetos correspondientes y los han insertado en su tapiz local de CAEAT. La salida del servicio, el atributo “a”, ha sido conectado a otros componentes locales de cada una de las máquinas. En el diagrama, la línea punteada alrededor de un componente indica que éste es remoto.

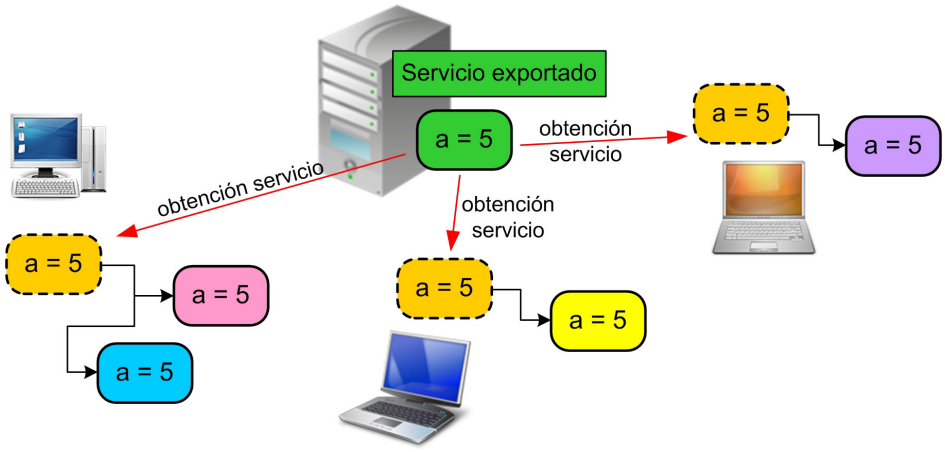

Figura 3.7: Utilización de un servicio por parte de tres clientes distintos

Con el transcurso del tiempo, un cliente puede decidir que se debe cambiar el valor del atributo. Si esto sucede, se lanzará, en el tapiz local del cliente que ha cambiado el atributo, un evento de cambio de valor. Este evento (`PropertyChangeEvent` del estándar *JavaBeans*) sirve para que todos los componentes conectados al que ha sufrido el cambio puedan actualizar el valor de su atributo.

Además, dado que se habrá invocado el método *get* del atributo, y éste habrá delegado en el método del mismo nombre del *proxy* del servicio, el objeto “real” exportado en el servidor también habrá tenido ocasión de actualizar su valor. Sin embargo, por el momento, la cadena de eventos termina ahí. Es necesario un mecanismo para que el servidor, bajo un cambio en uno de sus atributos, pueda notificar al resto de clientes que el valor de un atributo ha sido cambiado.

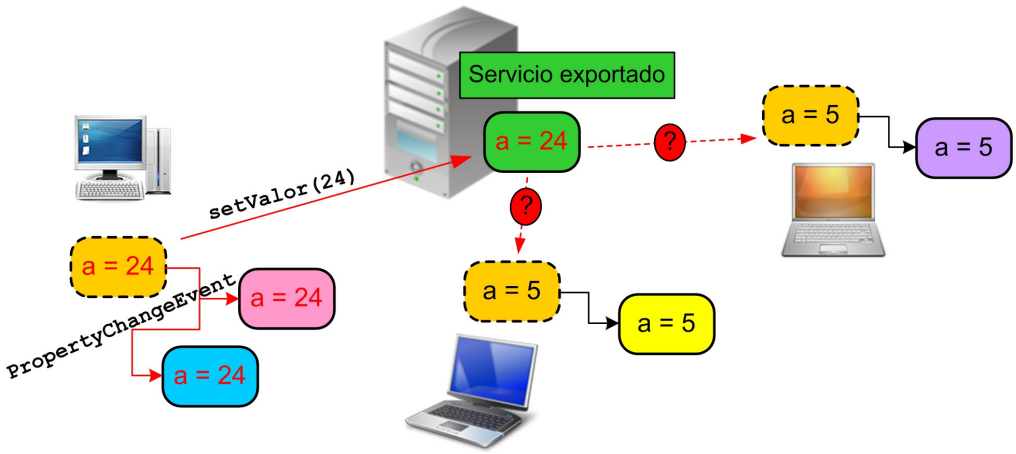

Figura 3.8: Problemática de la actualización automática del valor de las variables

3.4.2. Clases a utilizar

A continuación se describen algunas de las clases ofrecidas por las especificaciones de *Jini / Apache River* que permiten gestionar el lanzamiento y tratamiento de eventos remotos. Pese a que el entorno de programación ofrece más clases para poder desplegar esquemas de notificación más complejos, para el caso de CAEAT únicamente es necesario el uso de `RemoteEventListener`, `RemoteEvent` y `UnknownEventException`.

3.4.2.1. RemoteEventListener

```
public interface RemoteEventListener extends Remote, java.util.EventListener
{
        void notify(RemoteEvent theEvent)
                throws UnknownEventException, RemoteException;
}
```

Código 3. 8: Interfaz *RemoteEventListener*

Se trata de la interfaz principal de cualquier esquema de lanzamiento de eventos remotos. La clase que implemente esta interfaz será la receptora de los eventos. Obliga a la implementación de un único método `notify(RemoteEvent theEvent)`, que será llamado remotamente por las entidades generadoras de los eventos cuando éstas tengan algún cambio a notificar. El tratamiento del evento se debe implementar dentro de dicho método *notify* atendiendo al tipo de evento recibido, información que se encapsula en un objeto de tipo `RemoteEvent`.

3.4.2.2. RemoteEvent

```
public class RemoteEvent extends java.util.EventObject {
        public RemoteEvent(Object source,long eventID,
                long seqNum, MarshalledObject handback)
        public Object getSource () {...}
        public long getID() {...}
        public long getSequenceNumber() {...}
        public MarshalledObject getRegistrationObject() {...}
}
```

Código 3. 9: Clase *RemoteEvent*

Se trata del objeto que las entidades generadoras de los eventos pasan al/los receptor/es de los mismos mediante llamada al método remoto *notify*. Incluye campos para identificar unívocamente el

tipo de evento generado y su origen. Estos campos pueden ser usados por el programador para la implementación del esquema de notificación de eventos deseado.

- ***source***: el objeto generador del evento
- ***eventID***, ***seqNum***: números que ayudan a identificar el tipo de evento y su ordenación. La secuenciación de los eventos es importante ya que debido a que éstos viajan a través de la red pueden llegar al *listener* desordenados o incluso pueden llegar a perderse.
- ***handback***: objeto que encapsula información adicional sobre el evento. Este objeto es el que proporciona al programador mayor libertad a la hora de diseñar su esquema de notificación de eventos remotos, ya que las especificaciones no obligan a que deba seguir ninguna estructura concreta. La única condición, por ser un objeto de tipo `MarshalledObject`, es que sea apto para ser convertido a una secuencia de bytes y transmitido por la red. La estructura de este objeto, por lo tanto, será más o menos compleja en función del esquema de notificación que se haya diseñado.

3.4.2.3. UnknownEventException

```
public class UnknownEventException extends Exception {
    public UnknownEventException() {
        super();
    }
    public UnknownEventException(String reason){
        super(reason);
    }
}
```

Código 3.10: Clase *UnknownEventException*

Se trata sencillamente de la excepción que se debe lanzar si el objeto `RemoteEvent` recibido por el *listener* de eventos remotos no se reconoce como un evento válido.

3.4.3. Aplicación a CAEAT

El uso de las herramientas presentadas para la implementación de un esquema de notificación de eventos remotos en CAEAT se debe enfocar tanto desde el punto de vista de la propia plataforma como desde el de los servicios implementados.

Desde el punto de vista de los servicios, la clase *Server* (aquélla en la que tiene lugar el procesado real de los servicios) deberá guardar una lista actualizada con referencias a los clientes que están usando el servicio en cada momento. Dichas referencias serán objetos de la clase `RemoteEventListener` que se acaba de presentar.

Ante el cambio del valor de una propiedad por parte de un cliente, el *Server* deberá recorrer la lista de clientes y notificar a cada uno de ellos del cambio, mediante llamada remota al método *notify* presentado en el apartado anterior y mediante el envío de un objeto `RemoteEvent` convenientemente generado. El *Server* debe proporcionar también un mecanismo para que los clientes puedan darse de alta y de baja de la lista de *listeners*, que será usado en los instantes en que un cliente obtenga un nuevo servicio y deje de utilizarlo, respectivamente.

A continuación se presentan y detallan los fragmentos de código del *Server* relativos al lanzamiento de eventos remotos:

```
        private CopyOnWriteArrayList<RemoteEventListener> CAEATListeners;
        private Long eventSource;
        private long eventSequence;

        public long register(RemoteEventListener CAEATInstance) {
                if(!CAEATListeners.contains(CAEATInstance)) {
                        CAEATListeners.add(CAEATInstance);
                }
                return eventSource;
        }

        public void unregister(String nombreInstancia, RemoteEventListener rel) {
                lanzaEvento(0,nombreInstancia,null);
                CAEATListeners.remove(rel);
        }

        private void lanzaEvento(int tipoEvento, String info, Object[] objetos) {
                // 0 = Petición de desregistro
                // 1 = Actualización de alguna propiedad
                // 2 = Notificación de detención o caducidad de ese servicio
                // 3 = Cambio estructural en una agregación remota
                try {
                        MarshalledObject handback = new
                        MarshalledObject<SenetBeansNotification>(new
                        SenetBeansNotification(eventSource,info,objetos));
                } catch(IOException e) {e.printStackTrace(); }
                RemoteEvent ev = new
                RemoteEvent(this.getClass().getName(),tipoEvento,eventSequence,handback);
                eventSequence = ((eventSequence + 1) % Long.MAX_VALUE);
                notifyCAEAT(ev);
        }
```

Código 3.11: Funciones del servidor relativas al lanzamiento de eventos

El servidor guarda una lista llamada `CAEATListeners` que contiene referencias a todas las instancias de CAEAT que están haciendo uso del servicio en cada momento. Ofrece los métodos *register* y *unregister* para que CAEAT pueda registrarse como interesado en recibir los eventos del servicio ante la obtención del mismo; o pueda darse de baja de la lista de *listeners* en caso de su eliminación. Estos métodos son remotos, y accesibles a través del *proxy* del servicio.

Cada servicio define un número aleatorio `eventSource` que ayuda a los receptores de los eventos remotos a determinar unívocamente el servicio que está enviando el evento (si no se reconoce el remitente del evento se deberá lanzar una `UnknownEventException`). El servidor también lleva un control de la ordenación de los eventos lanzados mediante el atributo `eventSequence`, que se incrementa tras el lanzamiento de un evento.

El método más importante de la lógica de lanzamiento de eventos es *lanzaEvento*, que se ejecuta cada vez que ocurre una circunstancia que se deba notificar a los clientes. Además del cambio de valor en las propiedades, se lanzan eventos cuando suceden otras situaciones que serán explicadas con más detalle en capítulos posteriores. Se usa un número del 0 al 3 para codificar cada una de estas circunstancias, dejando la puerta abierta a la implementación de nuevos tipos de evento en el futuro.

Desde el punto de vista de CAEAT, el único requisito a cumplir es disponer de un módulo de software que actúe como `RemoteEventListener`. Se ha creado una clase que implementa dicha interfaz, y que realiza el tratamiento de los eventos adecuadamente en su método *notify*. Cada vez que se obtiene un nuevo servicio, es necesario llamar al método *register* a través del *proxy* del servicio, incluyendo la mencionada clase como parámetro. De la misma manera, tras la eliminación de un servicio el método *unregister* debe ser llamado para denotar explícitamente que la instancia de CAEAT ya no tiene interés en los futuros eventos remotos generados por el servicio.

Para una explicación más detallada y técnica del mecanismo de lanzamiento de eventos remotos desde los servidores, se aconseja la consulta de su código fuente en el anexo B, una vez se conozcan todas las posibles circunstancias que llevan al lanzamiento de eventos remotos.

3.5. CONCURRENCIA EN SERVICIOS REMOTOS

En un entorno distribuido como el que se ha descrito hasta el momento, únicamente resta por plantearse los problemas que pueden surgir a raíz del uso concurrente de los servicios por parte de un número indeterminado de clientes.

Se ha hecho especial hincapié en remarcar que las variables reales de proceso del servicio residen en el servidor, y que los clientes mantienen en su espacio local una copia sincronizada de dichas variables. Sin embargo, si las variables "reales" no se protegen contra el acceso concurrente, la sincronía se puede perder. Este apartado expone la problemática derivada de dicho acceso concurrente y la estrategia adoptada para evitarlo.

3.5.1. Descripción de la problemática

Para entender el problema de concurrencia presente en la arquitectura expuesta hasta el momento, se debe describir el funcionamiento de una llamada a un método remoto. En particular, hay dos características que ponen de relieve el problema:

- Cuando se realiza una llamada remota a través de un *Proxy*, el objeto remoto no se bloquea. Esto significa que el procesado que se realiza en la clase *Server*, que guarda las variables reales del servicio, no se realiza en exclusión mutua. Un número arbitrario de clientes podrían encontrarse, en un momento dado, llamando a la misma función remota.
- Según las especificaciones de RMI, cada llamada remota consume un *Thread* en la máquina servidora. Esto significa que por cada cliente que realiza una llamada remota a través de un *proxy*, la JVM receptora lanzará un *Thread* que ejecutará el código de la función y devolverá los resultados.

Tomando en consideración las dos características anteriores juntas, se llega a la conclusión de que, en un momento dado, se pueden tener en la máquina del servidor un número indeterminado de *Threads* activos que intenten trabajar sobre las mismas variables. Nótese que el problema de concurrencia se da únicamente en el servidor, no en los clientes: éstos se limitan a realizar llamadas a los métodos *getter* y *setter* a través de sus *proxies* en un ambiente totalmente *monothread*.

En un ambiente *multithread* como lo es pues el de los servidores, se debe garantizar la integridad de los atributos críticos. En tales entornos se puede dar la situación de tener diversos *threads* intentando leer / escribir en la misma variable. Dependiendo del orden en el que se ejecuten dichos *threads*, los lectores pueden obtener datos obsoletos (debido a que hay escritores cambiando el valor de la variable en ese mismo instante) o los escritores pueden llegar a solaparse entre ellos de manera que el valor final del dato no sea el correcto.

En general, el resultado de la ejecución de un proceso con problemas de concurrencia como los descritos es errático e imprevisible, y depende del orden en que la JVM decida que los *threads* deben de ser ejecutados, algo que escapa al control del programador.

3.5.2. Solución adoptada

Dado que el escenario que presentan la mayoría de servicios desarrollados para CAEAT es el de una serie de datos a los que se puede acceder mediante métodos *getter* y *setter*, se puede concluir (como ya avanzaba el razonamiento del apartado anterior) que se está ante un clásico problema de concurrencia entre lectores y escritores.

El escenario lectores / escritores es ampliamente recurrente y conocido en entornos de ejecución *multithread* como lo es el Lenguaje de Programación *Java*. Existen multitud de soluciones ampliamente documentadas para abordar problemáticas similares. En el caso de los servicios

desarrollados para CAEAT, se ha optado por la utilización de un monitor personalizado para proteger cada uno de los atributos críticos.

En Programación Orientada a Objetos, un monitor representa un objeto diseñado para ser usado de manera segura por más de un *thread*. La característica más útil de un monitor es que, en cualquier momento dado, como máximo un único *thread* estará ejecutando cualquiera de sus métodos. En otras palabras, los métodos de un monitor se ejecutan siempre en exclusión mutua.

Los monitores ofrecerán métodos *leer* y *escribir* que deberán ser invocados previamente a la lectura y escritura del atributo respectivamente. Internamente, el monitor dispone de una serie de variables de condición, que representan reglas que deben cumplirse para que el *Thread* que intenta acceder al recurso no sea frenado. Por ejemplo, si un escritor intenta escribir en el recurso, pero la variable de condición que permite la escritura no se cumple, dicho *Thread* escritor será puesto en espera hasta que la regla sea cierta.

A continuación se muestran extractos del código fuente del servidor que ilustran el uso de los monitores. Se debe notar que:

- Se usa un monitor específico para proteger cada una de las variables (no es posible protegerlas a todas mediante el uso de un único monitor).
- Como paso previo a la lectura de una variable, se debe obtener el "lock" del monitor; en otras palabras, se debe invocar el método `Leer()` para intentar que el monitor conceda acceso a la variable. Si no lo concede, el *Thread* que intenta leer restará bloqueado en la llamada a `Leer()` hasta que se lo concedan. Al terminar la lectura, el *Thread* debe invocar al método `salirLectura()` para notificar al monitor que ya ha acabado el proceso (en otras palabras, para liberar el "lock" que el monitor le había concedido).
- El razonamiento anterior es idéntico para el caso de las escrituras.
- En los procesos complejos que implican el manejo de más de una de las variables del servicio, se debe obtener el "lock" de todas ellas. Por ejemplo, para poder llamar a `realizaCuenta` se debe tener acceso de lectura al atributo mensaje, y acceso de escritura a los atributos `cuenta` y `letras`. No hay que olvidar liberar los monitores una vez finalizado el proceso.

```
private SenetBeansMonitor monMensaje;
private SenetBeansMonitor monCuenta;
private SenetBeansMonitor monLetras;

public String getMensaje() {
        String mensaje;
        monMensaje.Leer();
        mensaje = this.mensaje;
        monMensaje.salirLectura();
        return mensaje;
}

public void setMensaje(String mensaje) {
        monMensaje.Escribir();
        this.mensaje = mensaje;
        monMensaje.salirEscritura();
        lanzaEvento(1,"mensaje",null);
}
```

```
        public void realizaCuenta(boolean cuenta) {
                int nuevoLetras = 0;
                monMensaje.Leer();
                monCuenta.Escribir();
                monLetras.Escribir();
                if(cuenta) {
                        StringBuffer buff = new StringBuffer(mensaje);
                        for(int i=0 ; i<buff.length() ; i++) {
                                char c = buff.charAt(i);
                                if((c >= 65 && c <= 90) || (c >= 97 && c <= 122))
                                        nuevoLetras++;
                        }
                }
                this.cuenta = false;
                this.letras = nuevoLetras;
                monLetras.salirEscritura();
                monCuenta.salirEscritura();
                monMensaje.salirLectura();
                lanzaEvento(1,"letras",null);
        }
```

Código 3.12: Ejemplo de protección de las variables del servidor mediante monitores

Internamente, el monitor implementa su propia política de funcionamiento. Es decir, es el monitor quien decide de qué forma debe establecer las variables de condición para proteger a las propiedades siguiendo una filosofía determinada. Se debe hacer notar que el cambio de un tipo de monitor por otro no afecta al funcionamiento del servicio. Un monitor que implemente los cuatro métodos relacionados con la lectura y la escritura que se han expuesto anteriormente es un monitor apto para ser usado con los servicios desarrollados para CAEAT.

Para el presente proyecto, se ha optado por implementar un monitor llamado *SenetBeansMonitor* que actúa bajo las siguientes restricciones:

- La lectura de cualquier propiedad puede hacerse de manera concurrente. Un número ilimitado de *threads* podrían llegar a estar leyendo un dato al mismo tiempo.
- La llegada de un escritor debe frenar el flujo de lectores entrantes, para evitar que éstos monopolicen el recurso todo el tiempo.
- La escritura es la operación más delicada y se debe realizar en estricta exclusión mutua. Todos los lectores y escritores que intenten acceder al monitor mientras un escritor tiene su "lock" tomado, deberán aguardar a que la primera escritura termine.
- Por cada ejecución de un escritor, se permitirá entrar al monitor a todos los lectores que estaban esperando. Se evita así que un flujo continuo de escritores monopolicen el recurso.

El código que implementa este tipo de monitor, ampliamente comentado y razonado, puede consultarse en el anexo C.

4. CAEAT SERVICE MANAGER

Este capítulo describe la arquitectura y el funcionamiento de *CAEAT Service Manager* (CSM), un módulo de software complementario a la plataforma CAEAT pero independiente de ella, que sirve para el mantenimiento y gestión de los servicios publicados desde dicha plataforma. En primer lugar se presenta la problemática que origina la necesidad del diseño e implementación de este nuevo módulo de software. A continuación se detalla la solución adoptada, para a continuación realizar una exposición de las operaciones más importantes llevadas a cabo por CSM.

4.1. PROBLEMÁTICA

La principal razón de la existencia de un módulo de software independiente a CAEAT radica en la necesidad de disponer de una lógica que se responsabilice del mantenimiento de los servicios y garantice su supervivencia y disponibilidad hasta la llegada de su fecha de caducidad. Esto implica que los servicios publicados desde la plataforma CAEAT deben continuar operativos y atendiendo llamadas remotas de los clientes incluso después de que la ejecución de la plataforma CAEAT haya finalizado. La figura siguiente ejemplifica la situación que se acaba de describir:

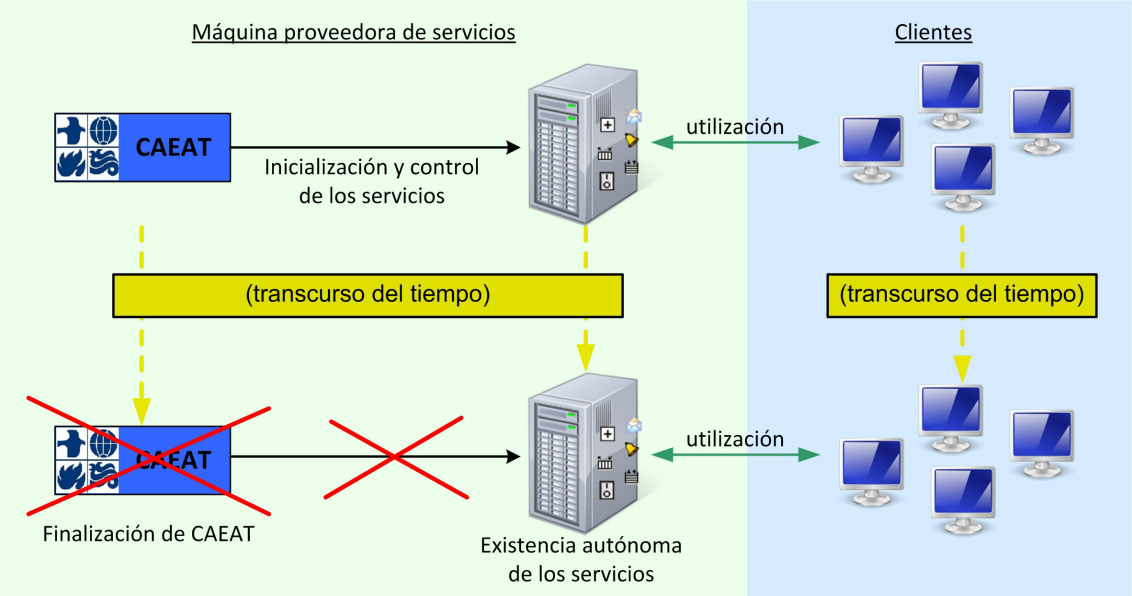

Figura 4. 1: Necesidad de una entidad externa a CAEAT que mantenga los servicios

En la ilustración anterior, las líneas punteadas amarillas representan el transcurso del tiempo. En la parte izquierda se representa la máquina que exporta y publica el servicio, mientras que en la parte derecha se representan los clientes que obtienen su *proxy* y hacen uso de él. Una instancia de CAEAT es ejecutada en la máquina servidor y se inician, exportan y publican una serie de servicios que son obtenidos por los clientes.

Con el transcurso del tiempo, la ejecución de CAEAT finaliza. El comportamiento deseable implicaría que los servicios continuasen activos en la máquina servidor y que los clientes pudiesen continuar haciendo uso de ellos hasta que no se detuviesen explícitamente o se alcanzara su fecha de caducidad. Sin embargo, si la lógica de publicación se ha llevado a cabo desde CAEAT, los servicios desaparecen junto con la plataforma.

El mantenimiento de un servicio activo implica la conservación del objeto *Servidor* que se ha exportado para la recepción de llamadas remotas. Además, se deben conservar también los objetos `Lease` retornados por los servidores de *Lookup* en el momento de la publicación de los objetos públicos del servicio. A través de dichos objetos se debe solicitar a los servidores LUS la renovación de la concesión de registro para evitar su desaparición de los servidores de forma prematura.

Se debe por lo tanto separar la lógica de edición y ensamblaje de las agregaciones y servicios (plataforma CAEAT) de la lógica de publicación y mantenimiento con vida de los servicios (módulo CSM). Esto implica que CAEAT y CSM deben ser ejecutados en máquinas virtuales de *Java* distintas. CAEAT debe poder ordenar la publicación de servicios a CSM, pero el mantenimiento de éstos debe recaer sobre el nuevo módulo de software para evitar que la finalización de la JVM que está ejecutando CAEAT provoque también la desaparición de los servicios.

4.2. SOLUCIÓN ADOPTADA

Este apartado expone los detalles de implementación de la solución adoptada, basada en la comunicación entre las dos máquinas virtuales de *Java* (la que ejecuta la plataforma CAEAT y la que ejecuta CSM), así como los principales objetos encapsulados por el CSM y su lógica de funcionamiento a alto nivel.

4.2.1. Comportamiento general de CSM

CAEAT Service Manager constituye un sencillo programa escrito en el lenguaje de programación *Java* completamente independiente a la plataforma CAEAT. Este programa es ejecutado por CAEAT únicamente en el momento en que se deseen publicar servicios, y su ejecución finaliza cuando se detienen (o caducan) todos los servicios que han sido publicados.

El lanzamiento automático de CSM desde CAEAT se consigue gracias a la función *exec* de la clase `Runtime` del lenguaje de programación *Java*. Esta función permite a cualquier aplicación *Java* la ejecución de procesos hijo. Es posible la ejecución de potencialmente cualquier aplicación si se dispone de los permisos para llevar a cabo dicha ejecución. Si se invoca al JRE (*Java Runtime Enviroment*), es posible ejecutar también otras aplicaciones *Java*. Esta última estrategia es la utilizada para el lanzamiento del CSM.

El proceso hijo restará activo incluso después de que el proceso padre que lo ha lanzado finalice su ejecución (la característica que se desea cumplir para evitar la problemática expuesta en el apartado anterior). El lenguaje *Java* permite incluso capturar la salida estándar y de error del proceso hijo para mostrarla por pantalla o redireccionarla hacia un fichero. El siguiente extracto de código muestra el lanzamiento de un proceso hijo y la redirección de sus mensajes de salida hacia la salida del proceso padre:

```
Process p;
(…)
try {
        p = Runtime.getRuntime().exec(command);
} catch(IOException e) { e.printStackTrace(); }

//Captura de la salida estándar del proceso que gobierna el CAEAT Service Manager:
Thread t_std = new Thread(new Runnable() {
        BufferedReader buff = new BufferedReader(new
                InputStreamReader(p.getInputStream()));
        public void run() {
                String line = null;
                try {
                        while ((line = buff.readLine()) != null) {
                                System.out.println("CSM | " + line);

                        }
                } catch(IOException e) { e.printStackTrace(); }
        }
});
t_std.start();
```

Código 4.1: Lanzamiento de un proceso hijo y captura de su salida estándar

El parámetro `command` que se proporciona a la función *exec* contiene el comando que se suministra al sistema operativo para el lanzamiento de la aplicación externa. Por ejemplo, para la ejecución de una aplicación *Java*, se debe invocar el JRE siguiendo la siguiente sintaxis: *java –cp <classpath> <clase principal>*, donde *<classpath>* es una lista ordenada de las localizaciones en las cuales reside el código fuente de la aplicación lanzada.

Una vez activo el CSM, éste no realiza ninguna acción: se limita a esperar de manera permanente la recepción de órdenes y llamadas por parte de la instancia de CAEAT que lo ha ejecutado. Típicamente, la primera de las solicitudes que recibirá el CSM será la de la publicación de un servicio, pero las funcionalidades finales de este módulo de software son mucho más diversas y se detallarán en los apartados siguientes. El proceso de CSM puede ser detenido bajo solicitud de CAEAT mediante interrupción de su *Thread* principal de ejecución (que se encuentra permanentemente en estado *sleep* a la espera de la recepción de llamadas por parte de CAEAT).

Los objetos principales manejados por el CSM para conseguir el mantenimiento y la supervivencia de los servicios gestionados son los siguientes:

- Conjunto de objetos *Publicador* de cada uno de los servicios publicados. Mediante estos objetos es posible el control del ciclo de vida de los servicios (véase capítulo 6). Resulta vital guardar una referencia a estos objetos ya que contienen el objeto exportado *Server*, que sería automáticamente des-exportado si ninguna JVM guardase una referencia a dicho objeto (y por lo tanto el servicio finalizaría abruptamente).
- Conjuntos de colecciones de objetos `Lease` que representan las concesiones obtenidas desde los servidores de *Lookup* para cada servicio.
- Listas con los tiempos de vida establecidos por el usuario para cada servicio. Las concesiones obtenidas por parte de los servidores LUS deben ser renovadas hasta alcanzar la duración del servicio deseada.
- Una colección de *Threads* que se encargan de la mencionada renovación. CSM mantiene un *Thread* para la supervisión de cada servicio publicado. Aunque pueda parecer que este escenario es ineficiente, en el capítulo 4.3 se expondrá cómo este conjunto de *Threads* se encuentran en un estado de *sleep* durante prácticamente el 100% del tiempo.

Los servicios mantenidos por CSM y sus objetos asociados están convenientemente indexados para poder diferenciarlos fácilmente y acceder a ellos con rapidez: cada servicio publicado tiene un índice único asociado que se utiliza para el almacenaje del resto de sus objetos en tablas ordenadas.

4.2.2. Comunicación entre máquinas virtuales de Java

La solución propuesta a la problemática de la supervivencia de los servicios implica que el proceso CAEAT deba ser capaz de realizar llamadas y solicitudes al proceso CSM, pasándole diferentes objetos como parámetro y recibiendo los resultados de vuelta. Debe haber, por lo tanto, una comunicación entre dos máquinas virtuales de *Java* que se ejecutan en la misma máquina física.

Dos procesos que se ejecutan en máquinas virtuales de *Java* distintas se comportan a todos los efectos de la misma manera que si se ejecutasen en máquinas físicas diferentes. Ello implica que para habilitar la comunicación entre ellas se debe acudir a alguno de los mecanismos tradicionales de comunicación entre entidades (*sockets*, envío de mensajes, etc.). Sin embargo, por proximidad con la temática del presente proyecto, se ha optado por la utilización de llamadas remotas mediante la utilización de las herramientas proporcionadas por el paquete *Java RMI*.

Esta solución es factible porque el escenario ofrece dos particularidades que la hacen propicia:

- La comunicación será unidireccional. El proceso de CAEAT realizará llamadas remotas al de CSM, pero nunca al contrario. Es propicio, por lo tanto, que CAEAT lleve a cabo dichas llamadas a través de un objeto *proxy* que represente a CSM.
- Dado que en la práctica ambos procesos se ejecutan en la misma máquina física y bajo el mismo entorno y estructura de clases, no se presenta la problemática expuesta en el capítulo 3.3.2 acerca de las llamadas remotas en RMI (*codebase*, servidor HTTP, resolución remota de clases, etc.).

Se ha utilizado el escenario RMI más sencillo posible: las clases `UnicastRemoteObject` y `LocateRegistry` permiten la exportación de un objeto para la recepción de llamadas remotas, generando una interfaz que puede registrarse en un puerto conocido de la máquina local. Otra máquina virtual puede, a posteriori, buscar y obtener dicha interfaz registrada en el puerto conocido para, a través de ella, realizar todas las llamadas remotas que le permitan controlar el proceso que ha sido exportado. Nótese la similitud con el proceso de exportación llevado a cabo por las librerías de *Jini / Apache River*.

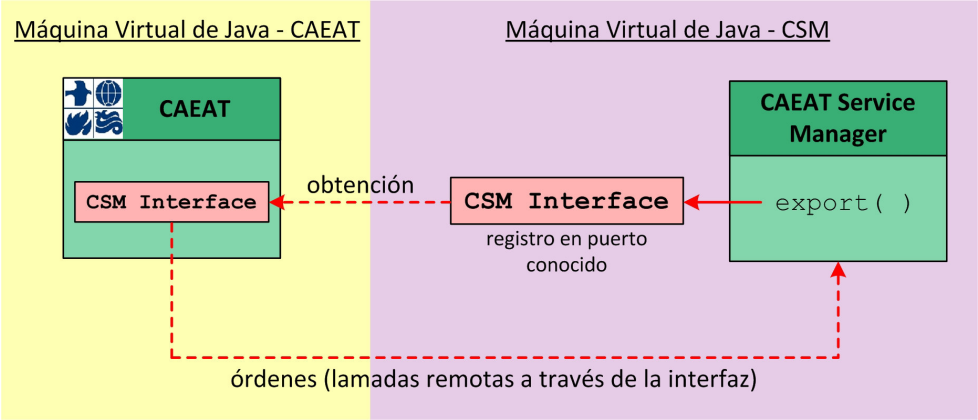

Figura 4.2: Esquema de comunicación entre CAEAT y CSM mediante llamadas remotas RMI

En el momento de la inicialización de CSM, éste registra su interfaz remota en un puerto conocido por ambas partes (CAEAT y CSM). Este puerto es determinista y elegible por el usuario mediante el cuadro de selección de opciones implementado para CAEAT. Tras realizar con éxito esta operación, la instancia de CAEAT puede buscar y obtener la interfaz registrada en dicho puerto, para a través de ella invocar los métodos que permiten controlar el CSM, proporcionándole los parámetros adecuados y recibiendo los resultados de vuelta.

Nótese que pese a ejecutarse ambos procesos en la misma máquina física, las llamadas que CAEAT realiza sobre el proceso CSM son llamadas remotas por tratarse de una comunicación entre máquinas virtuales de *Java* distintas. Esto implica que todos los objetos intercambiados durante la comunicación deben ser aptos para transmisión por la red, es decir, implementar la interfaz *Serializable* y ser totalmente aptos para serialización (tal y como se detalló en el capítulo 1.2.3.).

4.2.3. Lógica de inicialización

De las exposiciones anteriores se deduce que el proceso *CAEAT Service Manager* no está permanentemente activo, sino que es lanzado automáticamente desde CAEAT cuando se tiene la necesidad de publicar servicios, y es finalizado explícitamente cuando no quedan servicios que mantener. El objetivo principal de esta estrategia es mantener CSM activo únicamente si es necesario, y que los procesos de activación y finalización del proceso sean transparentes al usuario. Para ello, se ha implementado la siguiente lógica de inicialización:

1. **Inicialización de CAEAT**: se intenta buscar y obtener la interfaz remota de CSM.
    a. Si CSM ya estaba activo, se obtendrá su interfaz. Será posible visualizar y gestionar los servicios publicados, así como publicar nuevos.
    b. Si CSM no estaba activo, no se lleva a cabo ninguna acción. CSM no debe iniciarse innecesariamente, puesto que existe la posibilidad de que el usuario únicamente desee utilizar CAEAT como cliente o en modo local.
2. **Intento de publicación de un servicio**: se comprobará si existe en la máquina local un servidor HTTP activo y operando en el puerto correspondiente (requisito indispensable para poder desplegar servicios tal y como se expuso en el capítulo 3.3.2).
    a. Si no hay un servidor HTTP activo no se puede proceder con la publicación: se informa al usuario de que únicamente podrá utilizar CAEAT como cliente hasta que se inicie uno de estos servidores.
    b. Si hay un servidor HTTP activo pero no hay una instancia de CSM en ejecución, significa que el servicio bajo publicación es el primero que se publica en la máquina. Se lanzará CSM y se publicará el servicio mediante llamadas remotas a través de su interfaz. A partir de este momento, CSM restará activo de manera indefinida e independientemente de la existencia de CAEAT.
    c. Si hay un servidor HTTP activo y además CSM ya se encuentra operando, simplemente se publicará el servicio actual y se agregará éste al conjunto de servicios gestionados y mantenidos por CSM.
3. **Finalización de un servicio**: un servicio puede finalizar porque se alcanza la fecha de caducidad establecida o porque se finaliza de manera manual mediante la interfaz de CAEAT (véase capítulo 6 para más detalles).
    a. Si se trata del último servicio publicado desde esa máquina, CSM finaliza automáticamente para evitar el consumo innecesario de recursos.
    b. Si existen más servicios gestionados por CSM, se deben limpiar las variables relacionadas con el servicio detenido, pero CSM continúa operando de manera normal y manteniendo activos el resto de servicios.
4. **Finalización de CAEAT**: la plataforma CAEAT finaliza de manera normal en cualquier caso, haya o no servicios publicados en la máquina. Si los hay y CSM se encuentra activo, su proceso no es finalizado, sino que seguirá ejecutándose de manera silenciosa para mantener activos los servicios publicados.

4.3. OPERACIONES LLEVADAS A CABO

Se presentan en este apartado algunas de las operaciones más importantes llevadas a cabo por *CAEAT Service Manager*. En concreto se presentan las estrategias empleadas para conseguir la renovación periódica de las concesiones recibidas por parte de los servidores de *Lookup* y la publicación de servicios. Existen algunas operaciones de control de flujo de la vida de los servicios que se explican con más detalle en el capítulo 6, una vez expuestas éstas.

CAEAT Service Manager almacena todas las concesiones otorgadas por los servidores de *Lookup* a los objetos públicos de los servicios. En la práctica totalidad de los casos, las concesiones otorgadas por los servidores de Lookup *serán* mucho menores a la fecha de caducidad de los servicios. Es por ello que CSM debe solicitar periódicamente la renovación de dichas concesiones hasta que se alcance la fecha de caducidad seleccionada en el momento de la publicación.

CSM hace uso de un *Thread* específico para cada servicio con objeto de llevar a cabo la renovación periódica de las concesiones. Dado que la arquitectura *Jini / Apache River* prevé la necesidad de realizar renovaciones de grandes cantidades de objetos `Lease` para asegurar la supervivencia de los servicios, las API's ofrecen la clase `LeaseRenewalManager`, que ayuda a este tipo de operaciones. El algoritmo seguido por el *Thread* que se encarga del mantenimiento de un servicio se muestra en el siguiente diagrama:

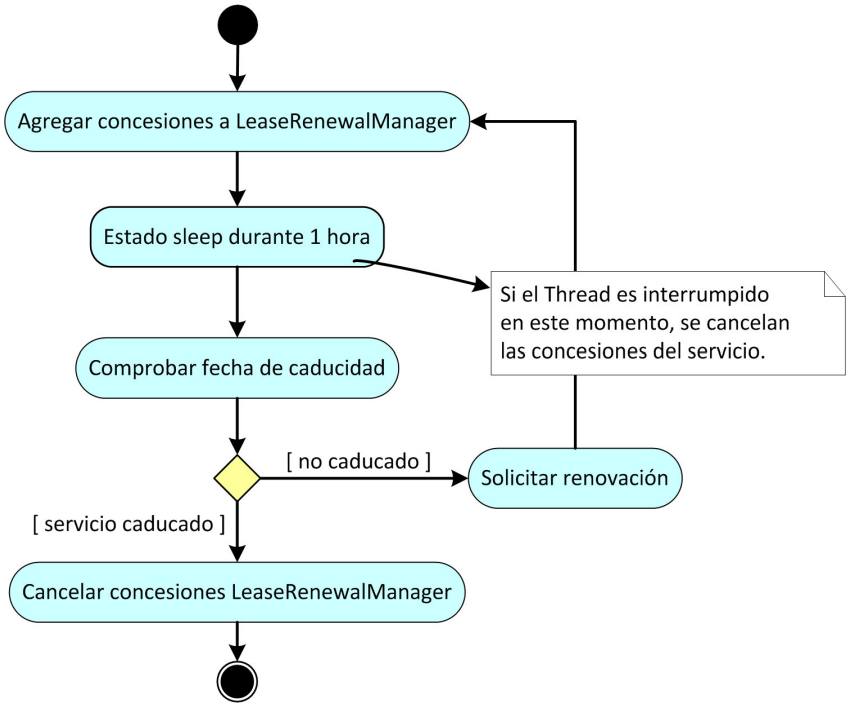

Figura 4.3: Algoritmo de renovación de las concesiones de los servicios

Cuando finaliza la publicación de un nuevo servicio, se lanza y almacena el *Thread* que se encargará de la renovación de todos los objetos `Lease` relacionados con dicho servicio. Todos estos objetos son agregados a una instancia de `LeaseRenewalManager`, que permite solicitar su renovación en conjunto y de una manera cómoda. A continuación el *Thread* entra en un bucle en el cual resta en estado de *sleep* la mayor parte del tiempo. En la implementación del presente proyecto se ha escogido como tiempo de *sleep* un valor de una hora debido a que los servidores de *Lookup* ofrecidos por las librerías de *Jini / Apache River* suelen proporcionar concesiones del orden de unas pocas horas (pese a depender el valor finalmente concedido de la carga de trabajo de dichos servidores).

Cada vez que el *Thread* renovador se despierta, comprueba si ha expirado la fecha de caducidad del servicio. La fecha de caducidad se encuentra almacenada en el CSM en valor absoluto en milisegundos, y únicamente se debe comparar con la fecha actual expresada en los mismos términos. Si el servicio ha caducado se procede a la cancelación de todos sus *leases* (tarea facilitada por el objeto de la clase `LeaseRenewalManager`). En caso contrario, se solicita la renovación de todos ellos (de nuevo de manera conjunta y rápida gracias a la mencionada clase).

Se debe destacar que todas las solicitudes de renovación de las concesiones que se realizan a los servidores de *Lookup* se llevan a cabo usando el parámetro `Lease.FOREVER`, es decir, solicitando un tiempo de registro indefinido. Nótese que el otorgamiento de concesiones por parte de los servidores LUS es una tarea *best-effort*; es decir, los servidores conceden un tiempo ostensiblemente inferior al solicitado basándose en la carga de trabajo que tengan en cada momento y en su implementación interna. Las instancias del servidor de *Lookup reggie*, utilizado durante la elaboración de este proyecto, suelen conceder tiempos máximos de 6 horas en ausencia de carga de trabajo. Este valor aumenta progresivamente junto con la carga de trabajo manejada por el servidor.

Pese a que la detención de servicios es un proceso tratado ampliamente en el capítulo 6, es remarcable en este contexto el hecho de que dicha detención provoca una interrupción al *Thread* renovador de las concesiones, que procede a la cancelación de éstas de la misma manera que si el servicio hubiese caducado.

La otra gran operación que lleva a cabo CSM es la publicación de servicios. Dicho proceso se puede resumir con el siguiente diagrama de secuencia:

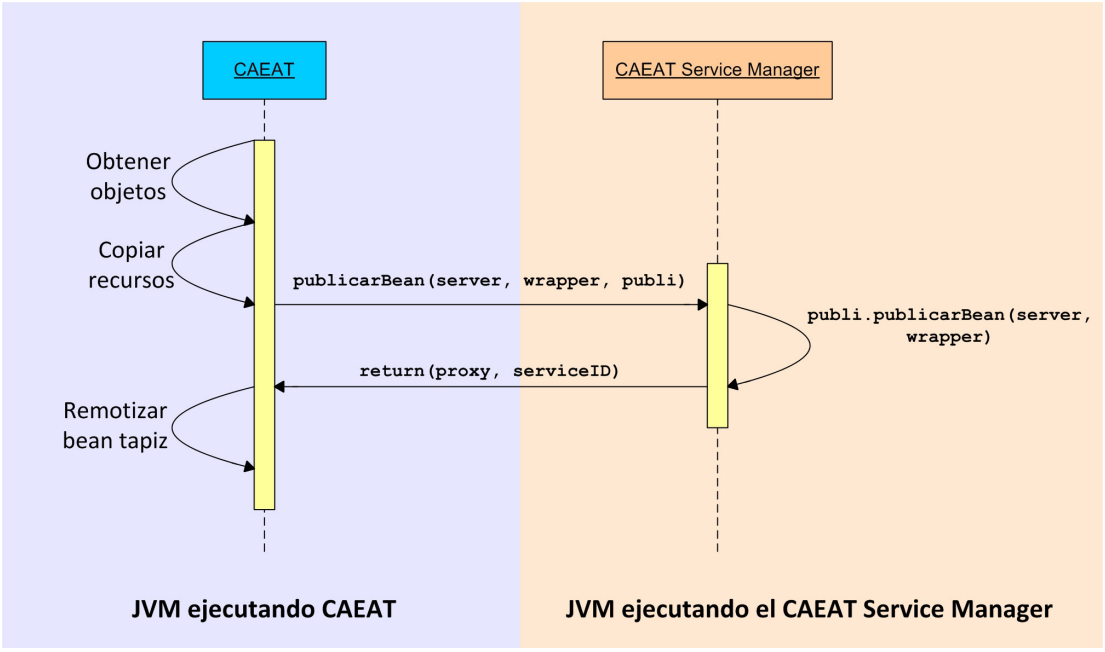

Figura 4.4: Proceso de publicación de un servicio simple (*bean*)

CAEAT obtiene los objetos implicados en la publicación del servicio (servidor, *proxy* y *wrapper*) y copia sus definiciones de clase (archivos de formato *class*) a la carpeta pública del servidor HTTP para hacerlas accesibles a los clientes. A continuación ordena a CSM (mediante llamada remota a través de su interfaz) la publicación del servicio simple, proporcionándole los mencionados objetos como parámetro.

En el capítulo 3 se expuso que el diseño de los servicios se ha realizado de tal manera que es posible invocar una función llamada `publicarBean` sobre el objeto publicador del servicio y la publicación se lleva a cabo haciendo abstracción de los detalles de ésta. CSM almacena una referencia al objeto *Publicador* del servicio, del cual extrae la colección de objetos `Lease` obtenidos por parte de los LUS, devolviendo al mismo tiempo unos resultados a CAEAT que son utilizados para convertir a servicio remoto el componente correspondiente del tapiz.

Este proceso se itera tantas veces como sea necesario en caso de la publicación de una agregación de componentes (se detallará el proceso en el capítulo siguiente). La publicación de servicios especiales como los expuestos en el capítulo 8 sigue un esquema muy similar.

4.4. CLASSLOADERS

Una vez expuestas la naturaleza y funcionalidades brindadas por CSM, este apartado realiza un inciso para detallar los distintos *ClassLoaders* presentes en la plataforma CAEAT. La razón principal que lleva a realizar esta exposición aquí, radica en que CSM hace uso de un *ClassLoader* customizado para solucionar problemáticas que se exponen a continuación. Previamente, se detallan las funciones de los objetos de tipo *ClassLoader* en el lenguaje de programación *Java*.

4.4.1. Función de los ClassLoaders

Una aplicación de la plataforma *Java* consta principalmente de un conjunto de archivos binarios que representan las clases que forman parte de dicha aplicación. Estos archivos tienen la terminación *class* y son el resultado de la compilación del código fuente de la aplicación desarrollada. En el momento de la ejecución de la aplicación, la Máquina Virtual de *Java* lee dichos ficheros de código binario como una ristra de *bytes* que es cargada en memoria y resuelta (si es posible) en una clase válida y utilizable (instanciable en forma de objetos concretos) durante el ciclo de vida de la aplicación.

La operación anterior, denominada carga y definición de clases, es llevada a cabo por clases derivadas de la clase `ClassLoader`, un elemento básico en la arquitectura de la plataforma *Java*. La plataforma *Java* incluye un *ClassLoader* por defecto al que se delega la carga de la mayoría de clases que forman la aplicación, pero existen multitud de tipos de *ClassLoader* derivados de éste que ofrecen un gran abanico de opciones adicionales al proceso de carga y definición de clases.

Dado que el proceso de carga y definición de una clase siempre acaba desembocando en la conversión de una ristra de *bytes* que representa a la clase en un objeto utilizable, dicha información puede proceder de localizaciones muy diversas: el sistema de ficheros de la máquina que ejecuta la aplicación, un archivo de tipo *jar* que se agrega a posteriori y contiene una colección de nuevas clases, o una localización remota en otra máquina. La plataforma *Java* ofrece diferentes tipos de *ClassLoader* para afrontar todos los posibles casos.

4.4.2. ClassLoaders utilizados en CAEAT

A continuación se describen los tres tipos de cargadores de clases utilizados en la plataforma CAEAT:

- ***System ClassLoader***: el cargador de clases por defecto de la máquina virtual, que se encarga de la carga de todas las clases que se encuentran disponibles en el entorno local de la aplicación (*classpath*) en el momento de iniciarse ésta. Es posible obtener una referencia a dicho cargador mediante la llamada estática *ClassLoader.getSystemClassLoader()*.

- ***CargadorClasesJar (extends URLClassLoader)***: la clase `URLClassLoader` sirve para facilitar la carga de clases a partir de una URL que apunta a su localización (ya sea una localización remota o local). `CargadorClasesJar` hereda de dicha clase y permite la carga de las clases contenidas en los archivos *jar* que constituyen librerías de nuevos componentes y que se añaden de manera dinámica y en tiempo de ejecución a la plataforma.

  Cada librería añadida cuenta con un *CargadorClasesJar* asociado que mantiene una URL apuntando al archivo *jar* que contiene las nuevas clases de la librería. La razón que lleva a heredar de `URLClassLoader` en lugar de usarlo tal y como es, radica en el enriquecimiento que se opera en el cargador proporcionándole operaciones adicionales que permiten controlar de manera más precisa la carga de las clases que forman la librería. Una de estas operaciones consiste en el descifrado de las clases de las librerías encriptadas, que se detallará en el capítulo 8.

- ***CargadorClasesRemotas (extends PreferredClassProvider)***: el cargador de clases llamado `RMIClassLoaderSpi` o el cargador `PreferredClassProvider`, incluidos en las librerías de *Jini / Apache River*, deben ser configurados como cargadores por defecto en toda aplicación que se disponga a exportar y publicar objetos y aceptar llamadas remotas sobre ellos. Esto es necesario, entre otros motivos, para que la anotación de la propiedad *codebase* se realice correctamente sobre los objetos remotos.

Tal y como se explicó en el capítulo 3.3.2, todo objeto susceptible de ser obtenido por clientes remotos es anotado con la propiedad *codebase*, que indica la localización original desde la que es posible obtener la definición de la clase. Esta operación se realiza de manera automática si el cargador establecido es uno de los dos especificados. De nuevo, la razón del establecimiento de un cargador que hereda de `PreferredClassProvider` (y de la no utilización directa de éste) se debe al intento de controlar y enriquecer la manera en que las clases son cargadas (adaptando algunas rutas y nombres para evitar errores en el formato de las URL's manejadas por sistemas operativos distintos).

4.4.3. Motivación de uso de CargadorClasesRemotas

El establecimiento de `CargadorClasesRemotas` como el cargador de clases por defecto de CSM viene motivado por la necesidad de cambiar explícitamente la anotación del *codebase* de los objetos que se publican. En el lenguaje *Java*, por defecto, la *codebase annotation* establecida en un objeto que se serializa y se transmite por la red permanece inmutable a lo largo de todos los "saltos" que dicho objeto pueda realizar durante su ciclo de vida.

La *codebase annotation* es establecida y consultada mediante el cargador de clases que se ha hecho cargo de resolver la clase una de cuyas instancias resulta ser el objeto serializado y transmitido. En la plataforma CAEAT, los objetos a publicar provienen de los archivos *jar* que contienen las librerías de componentes, por lo que están cargados por `CargadorClasesJar`. Las especificaciones de `URLClassLoader` estipulan que en este caso los objetos sean anotados con la URL encapsulada por dicho cargador, que tal y como se ha detallado apunta al archivo *jar* de la máquina local.

En la siguiente figura se ilustra este comportamiento: la entidad de la izquierda representa la instancia de CAEAT que publica los objetos. Estos objetos son anotados con un *codebase* que apunta al archivo *jar* de la librería, ubicado en el sistema de ficheros de la máquina local (por ejemplo, en la unidad local "C:"). Este comportamiento es útil debido a que CSM no tiene en su *classpath* los archivos *jar* de las librerías, y por lo tanto los objetos recibidos desde CAEAT le son desconocidos. Sin embargo, puede fácilmente acceder a la definición de la clase (el archivo *class*) de los objetos a través de dicha ruta.

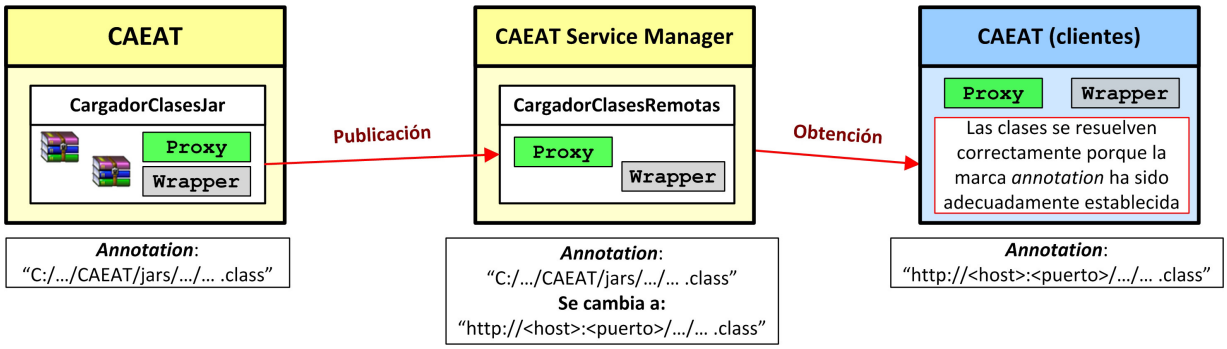

Figura 4.5: Utilización de *CargadorClasesRemotas* para el adecuado establecimiento del *codebase*

Sin embargo, los objetos no se pueden publicar en este estado, puesto que en recepción se consultará su *codebase annotation* y, de encontrarse ésta establecida a la ruta local del archivo *jar*, el cliente no será capaz de encontrar las definiciones de clase. Es por ello que se enriquece la clase `PreferredClassProvider` para el establecimiento manual de esta propiedad a un valor propicio para la plataforma, naciendo de esta manera `CargadorClasesRemotas`. Dicho "valor propicio" consiste en la URL en la cual el servidor HTTP se encuentra sirviendo las clases a los potenciales clientes, y a través de la cual dichas clases son visibles desde el exterior.

4.5. INTERFAZ DEL GESTOR DE SERVICIOS

Se ha dotado a la interfaz de usuario de CAEAT de la posibilidad de visualizar los servicios gestionados en un momento determinado por la instancia de *CAEAT Service Manager* que está ejecutándose en la máquina local (si la hay). Para ello, se ha habilitado una entrada de menú que muestra al usuario la interfaz que se presenta a continuación:

Figura 4.6: Interfaz del gestor de servicios publicados

Dicha interfaz consulta a CSM el número de servicios publicados en ese momento y, para cada uno de ellos, muestra información útil como su nombre, su icono, su tipología, su estado, etc. Esta interfaz gestora de servicios permite llevar a cabo algunas operaciones de control de flujo de la vida de los servicios tales como su detención o su des-publicación. Estas operaciones especiales de control serán detalladas en el capítulo 6.

Si no hubiese servicios desplegados en la máquina en el momento de la consulta a la interfaz del gestor de servicios, significaría que CSM no se está ejecutando. Este hecho es detectado y el usuario es informado de que no existen servicios publicados desde la máquina.

Figura 4.7: Interfaz mostrada en caso de ausencia de servicios publicados

5. AGREGACIONES REMOTAS DE COMPONENTES

Hasta este momento se ha hablado de "servicios remotos" como las simples e individuales piezas de software que se presentaron en el capítulo 1. Estos sencillos componentes software, que en el entorno de CAEAT reciben el nombre de *beans* por seguir la filosofía de funcionamiento del estándar *JavaBeans*, representan utilidades que realizan una función muy concreta bajo demanda. En el capítulo 2 se presentaron las librerías de componentes desarrolladas para CAEAT, que incluían componentes capaces de realizar operaciones matemáticas, comparaciones, adquisición de *inputs* de usuario mediante componentes gráficos, etc.

En los capítulos 3 y 4 se ha presentado la arquitectura diseñada para conseguir que estos componentes presten su servicio de un modo distribuido, es decir, que cualquier máquina de la red pueda adquirirlos, utilizarlos mediante llamadas remotas a sus métodos y recoger los resultados. Sin embargo, no se ha hecho referencia al encaje entre la distribución de los servicios y el montaje y edición de agregaciones de componentes, objetivo principal del presente proyecto.

CAEAT es una herramienta de montaje, agregación y edición de componentes software sencillos bajo interfaz gráfica. Se debe extender el concepto de "agregación" al entorno distribuido y orientado a servicios que se ha implementado hasta el momento. Este capítulo aborda toda la lógica que rodea a la creación, publicación, edición, etc. de agregaciones compuestas por "servicios", en el sentido más amplio de la palabra: éstos podrán ser locales, podrán ser obtenidos remotamente o podrán ser una combinación de los dos anteriores.

5.1. CONCEPTO DE "AGREGACIÓN REMOTA"

Como ya se expuso en el capítulo 2, una agregación de componentes en el marco de CAEAT consiste en la interconexión de componentes software sencillos. Esta interconexión se realiza de manera visual mediante conexiones en el tapiz de CAEAT, pero internamente los componentes se registran los unos a los otros como *listeners* de eventos de cambios de propiedades. De esta manera, la red funciona en base al lanzamiento y la propagación de eventos entre componentes cuando ha sucedido un cambio en algún componente de la red.

Los componentes de las agregaciones locales que se presentaron en el capítulo 2 provenían de librerías de componentes: recursos encapsulados en archivos *jar* que era posible añadir a la plataforma en tiempo de ejecución, y que eran listados en una paleta para su cómoda inserción en el tapiz de CAEAT. El objetivo de la adaptación a remoto de las redes de componentes es poder llegar a montar unas agregaciones compuestas por componentes de naturaleza muy diversa:

- Componentes locales provinentes de una librería (los expuestos en el capítulo 2)
- Componentes remotos que se están ejecutando en otra máquina (no importa cuál) o incluso en la máquina propia (no se debe hacer ningún tipo de distinción)
- Otras agregaciones de componentes remotos, cuyos servicios internos pueden estar "viviendo" en multitud de máquinas distintas a lo largo de la red.

Obviamente, los componentes y agregaciones locales deben poder ser conectados con los componentes y agregaciones remotos. La propagación de las propiedades de los servicios debe ser coherente tanto si se trata de servicios locales como remotos. A nivel del tapiz de la plataforma CAEAT, no debe haber distinción entre la utilización de servicios locales o remotos.

En la siguiente figura se muestra el aspecto de una agregación con presencia de componentes de naturaleza muy diversa. Obsérvese el cambio general en el aspecto visual de los componentes, así como la iconografía empleada en los componentes remotos, que será detallada en apartados posteriores.

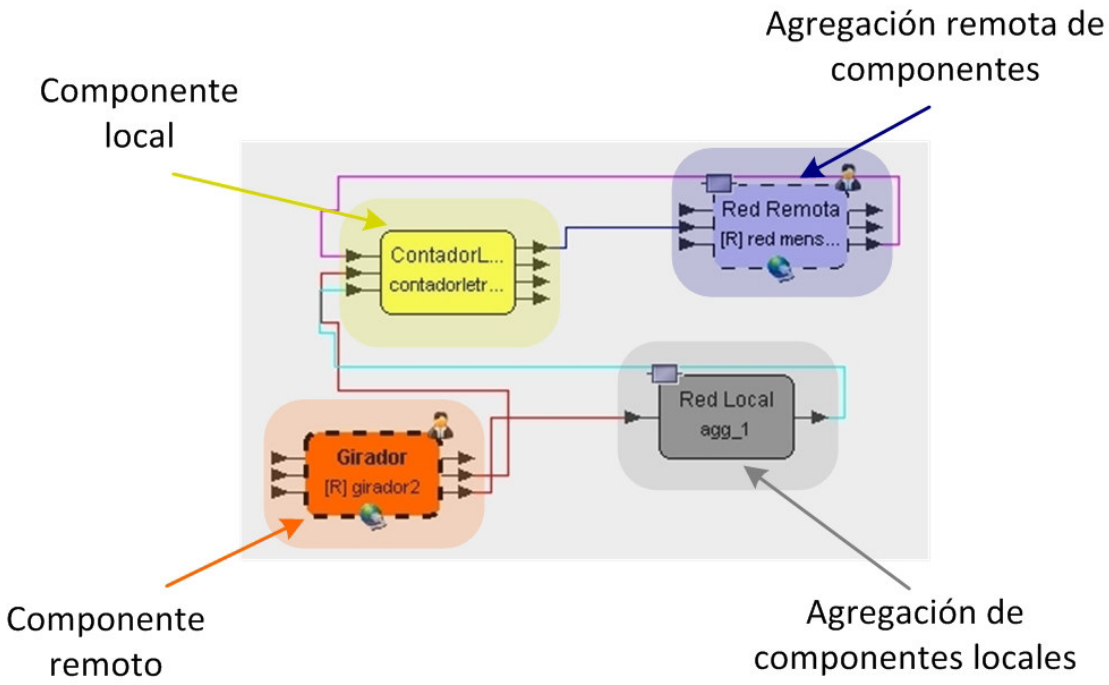

Figura 5.1: Agregación remota de componentes

La red incluye algunos componentes que realizan funciones de tratamiento sobre cadenas de caracteres, como un Girador de mensajes o un Contador de Letras. Estos componentes se han elaborado durante el presente proyecto en la fase de desarrollo para poder incorporar progresivamente nuevas funcionalidades a los componentes locales sin riesgo a destruir el buen funcionamiento de las librerías que ya se encontraban implementadas.

El diseño del lanzamiento de eventos remotos para mantener actualizados los valores de las propiedades de un servicio (expuesto en el capítulo 3.4) se realizó con la finalidad de garantizar la coherencia de las propiedades en agregaciones como la mostrada en la figura anterior:

- Si se produce un cambio de valor en alguna propiedad de un componente local (por ejemplo, el Contador de Letras de la figura anterior), éste lanzará un evento a sus *listeners* del tapiz de CAEAT (es decir, a los componentes que están directamente conectados a él).
- Si se produce un cambio de valor en alguna propiedad de un componente remoto (por ejemplo, el Girador de la figura anterior), éste también lanzará un evento a sus "vecinos" del tapiz. Sin embargo, por tratarse de un componente remoto, también propagará el nuevo valor del atributo hacia el servidor a través de su *proxy*, de manera que las variables "reales" del servicio queden actualizadas.
- Si un componente remoto recibe por parte del servidor un evento de notificación de cambio de valor de una propiedad, éste propagará dicho cambio a los componentes "vecinos" del tapiz pero, obviamente, no se lanzará evento remoto hacia el servidor.

Cabe destacar que el cambio de valor de una propiedad puede provenir de una acción de usuario o de un evento (ya sea local o remoto) que notifique dicho cambio. Se debe recordar asimismo que la notificación de cambio de valor en el entorno local (a los componentes interconectados entre sí) se realiza mediante la clase `PropertyChangeEvent` del estándar *JavaBeans*, mientras que los eventos provinentes del servidor se encapsulan en un objeto de la clase `RemoteEvent` del entorno *Jini / Apache River*.

Se consigue mediante esta metodología la total compatibilidad entre servicios locales y remotos y la coherencia en el valor de las propiedades tanto a nivel local como a lo largo de toda la red.

5.2. PUBLICACIÓN DE UNA AGREGACIÓN

Tal y como si se tratase de servicios simples, las agregaciones de componentes construidas mediante CAEAT deben poder ser exportables y publicables "como un todo" para su uso por parte de los clientes de la red. Los clientes deben poder obtener una agregación e insertarla en su tapiz para disfrutar de sus servicios, como si ésta fuese cualquier otro servicio local tomado desde la paleta.

A nivel de la plataforma CAEAT, no debe haber distinción funcional entre una agregación remota y cualquier otro componente. La agregación remota definirá una interfaz con los clientes que determinará qué propiedades quedan expuestas a éstos y, por lo tanto, qué métodos y funcionalidades de la agregación pueden ser llamados. La agregación, por compleja que sea, será vista desde fuera como una "caja negra" más, que deja expuestas una serie de propiedades. En la mayoría de casos, el cliente que haga uso de la agregación únicamente la conocerá "desde el exterior", sin llegar a saber (ya que no le es necesario) cómo está estructurada por dentro o qué componentes internos utiliza. Hay excepciones a esta regla que se matizan más adelante.

Este apartado detalla la estrategia adoptada para conseguir la publicación de agregaciones de componentes, así como las adaptaciones que se han realizado a la interfaz de usuario de CAEAT para permitir, en el momento de la publicación, la elección de los datos relevantes sobre la agregación.

5.2.1. Estrategia general de publicación

Una agregación no es otra cosa que la suma de un número determinado de componentes o servicios simples. Partiendo de esta base, la metodología de publicación implementada pasa por la publicación individual y secuencial de todos y cada uno de los servicios que forman la agregación. Todos los sub-servicios que forman una agregación remota deben disponer de sus objetos públicos (el *Proxy* y el *Wrapper*) publicados en los servidores de *Lookup* de manera que los clientes que deseen hacer uso de la agregación puedan obtenerlos.

Sin embargo, el proceso no puede acabar ahí, puesto que es necesario algún otro objeto que defina a la agregación como tal. De cara a los usuarios, una agregación formada por N sub-servicios debe ser vista como un único componente. En ningún caso se le presentará al usuario los N sub-servicios de manera individual, y obviamente, en ningún caso se les permitirá tomar y utilizar alguno de los N sub-servicios individualmente (la agregación es un todo irrompible).

Por ello, se ha optado por generar un documento XML que describa la estructura de la agregación y que sea de dominio público, pudiendo ser usado por los clientes para reconstruir la red. En el capítulo 2 se explicó que durante el guardado de agregaciones locales en ficheros *aec* se generaba un documento XML con un formato bien definido que permitía reconstruir la estructura de la red cuando ésta era cargada. Durante el proceso de publicación se genera una versión de dicho documento adaptado al caso de las agregaciones remotas. El formato de este documento se muestra en un apartado posterior.

Asimismo, se han creado las clases `AggregationProxy` y `AggregationServer`, que imitan el comportamiento de cualquier otro servicio.

- **`AggregationServer`**: es una clase que se exporta como servicio. La única propiedad contenida en este servicio es el documento XML descriptivo de la agregación. Como se detallará en los apartados siguientes, los usuarios de la agregación dispondrán de una copia local del XML de la red, mientras que el documento real residirá en el servidor.

  Dado que bajo ciertas circunstancias los usuarios podrán modificar la agregación, y por lo tanto generar un nuevo XML, el documento ofrece métodos *get* y *set* como cualquier otra propiedad de servicio, y es protegido contra el acceso concurrente de la misma manera (ver capítulo 3.5, acerca de la concurrencia en el lado del servidor).

- **`AggregationProxy`**: es el objeto que se publica en los servidores LUS para dar a conocer la agregación a la red, y que actúa de puente para poder obtener y modificar su documento XML. Es el único objeto perteneciente a la agregación que será visible desde CAEAT.

  Como se explicó en el capítulo 3.3, los servicios visibles por CAEAT en una búsqueda se marcan con el campo "CAEAT_Service" como dato de *Entry* en el momento del registro. Los objetos relativos a los sub-servicios de una agregación no se marcan de esta manera, y son ignorados durante una búsqueda. Se consigue de esta manera que una agregación de N sub-servicios sea vista únicamente como un "bloque" desde el exterior, y se garantiza que ninguno de sus N sub-servicios sean accesibles de manera individual.

5.2.2. Interfaz de publicación

En la figura siguiente se muestra y se desglosa la interfaz de usuario que se ha implementado para permitir la elección de datos relevantes acerca de la agregación en el momento de la publicación. Algunas de las características de las que se puede dotar a una agregación remota son explicadas en más detalle en capítulos posteriores.

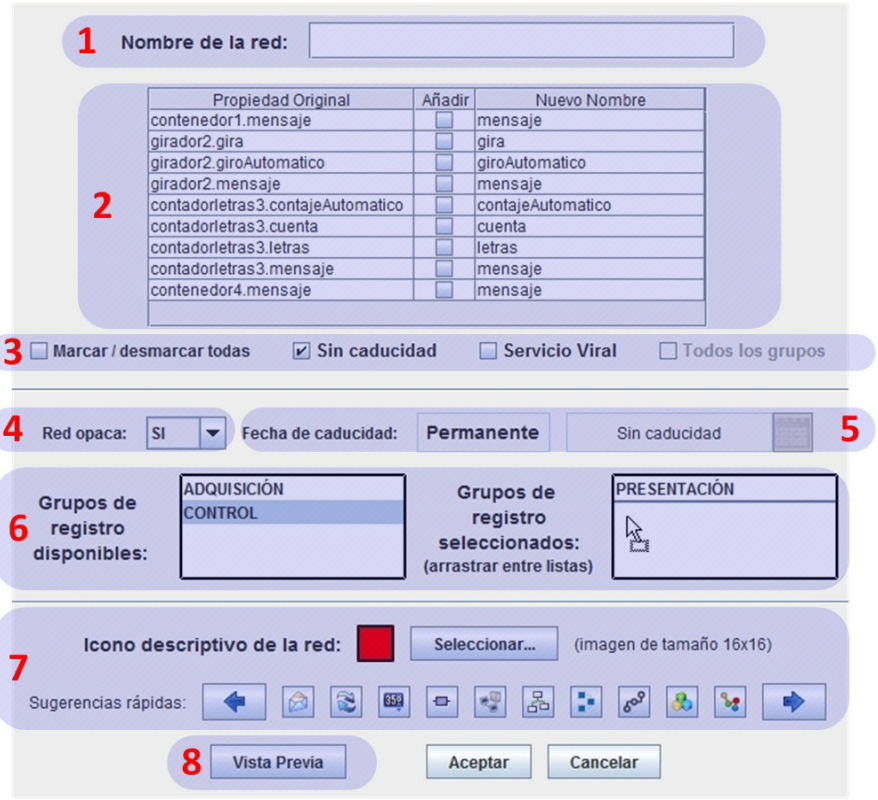

Figura 5.2: Desglose de la interfaz de publicación de agregaciones

1 – Nombre: se permite escoger un nombre para la red. Este nombre se guardará en su documento XML descriptor y servirá para mostrarlo al usuario como resultado de las búsquedas así como en otras interfaces gráficas.

2 – Selección de patillas: se deben seleccionar aquellas propiedades de la agregación bajo publicación que se desea que sean visibles desde el exterior. Con este paso se está definiendo la interfaz que la agregación ofrecerá al resto del mundo (las patillas que serán accesibles desde fuera de la "caja negra"). Es obligatoria la elección de una patilla como mínimo (ya que la publicación de una agregación sin interfaz externa no tiene sentido), y dado que las propiedades externas no pueden contener nombres repetidos, se ofrece la posibilidad de editarlos.

3 – *Checkboxes*: se ofrecen un *checkbox* para facilitar el marcado / desmarcado de todas las filas de la tabla anterior. Se ofrece asimismo la posibilidad de no elegir fecha de caducidad para el servicio (se renovará su concesión indefinidamente), siendo ésta la opción por defecto. El concepto de "servicio viral" se explicará en el capítulo 6.

4 – Opacidad: se permite seleccionar el estado de opacidad de la red, concepto que se explica más adelante en este mismo capítulo.

5 – Fecha de caducidad: si el *checkbox* "Sin caducidad" está activo, el servicio será renovado permanentemente. En caso contrario, es necesario elegir una fecha de caducidad. La unidad mínima temporal para la fecha caducidad de los servicios es de un día. Se ofrece un calendario emergente para facilitar la selección de la fecha, así como información sobre el tiempo de vida del servicio, calculado a partir de la fecha actual.

Figura 5.3: Calendario emergente de selección de fecha de caducidad

6 – Grupos de registro: se debe elegir como mínimo un grupo de *Lookup* al que la agregación pasará a formar parte. La lista del lado izquierdo se rellena de manera dinámica realizando un sondeo a los servidores de *Lookup* de la red para determinar cuántos grupos de registro distintos hay presentes en el momento de la publicación. Los grupos deseados se deben arrastrar a la lista del lado derecho mediante una interfaz *drag & drop*.

7 – Icono descriptivo: es posible asociar un icono de tamaño 16x16 píxeles a la agregación. Este icono será mostrado en la paleta de los clientes al realizar una búsqueda, y en otras ventanas al realizar ciertas operaciones. Se ofrecen algunos iconos por defecto, entre los que se incluyen los iconos descriptivos de los componentes que forman la agregación, pero también se ofrece la libertad de poder adjuntar un icono desde el sistema de ficheros de la máquina.

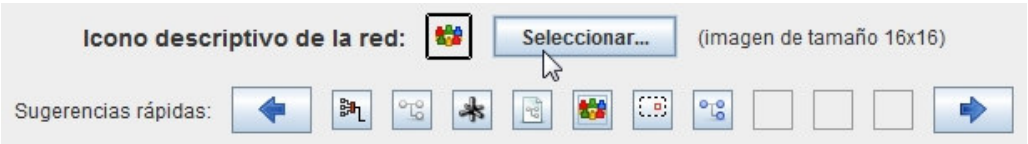

Figura 5.4: Selección del icono descriptivo de la agregación

El icono seleccionado pasa a ocupar la casilla anteriormente roja. Si no se elige icono, se escoge uno al azar de entre los propuestos. Dado que el icono es un recurso que el servicio debe ofrecer a los clientes, se copiará a la carpeta pública del servidor HTTP para que los clientes puedan obtenerlo junto a los archivos *class* que conforman las clases del servicio.

8 – Vista previa: tras la selección de patillas y nombre de la agregación, es posible realizar una previsualización de su interfaz externa. Es decir, se mostrará al usuario el modo en que la agregación será vista "desde fuera".

5.2.2.1. Publicación de un único componente

Se puede dar el caso particular de querer publicar una red formada por un único componente. En este caso, el componente se publica como un servicio autónomo, y no se llega a generar ningún documento XML ni se publica ningún *proxy* de la agregación. Puesto que tampoco será necesaria la elección de una interfaz externa (ya que dicha interfaz externa la constituirán las propias patillas del componente), la interfaz de publicación es ligeramente más simple:

Figura 5.5: Interfaz simplificada para el registro de servicios simples

5.2.3. Proceso de publicación de la agregación

Se detalla en este apartado el proceso que lleva a cabo CAEAT para conseguir la publicación de una agregación. El proceso puede resumirse en el siguiente diagrama, que se desglosa a continuación:

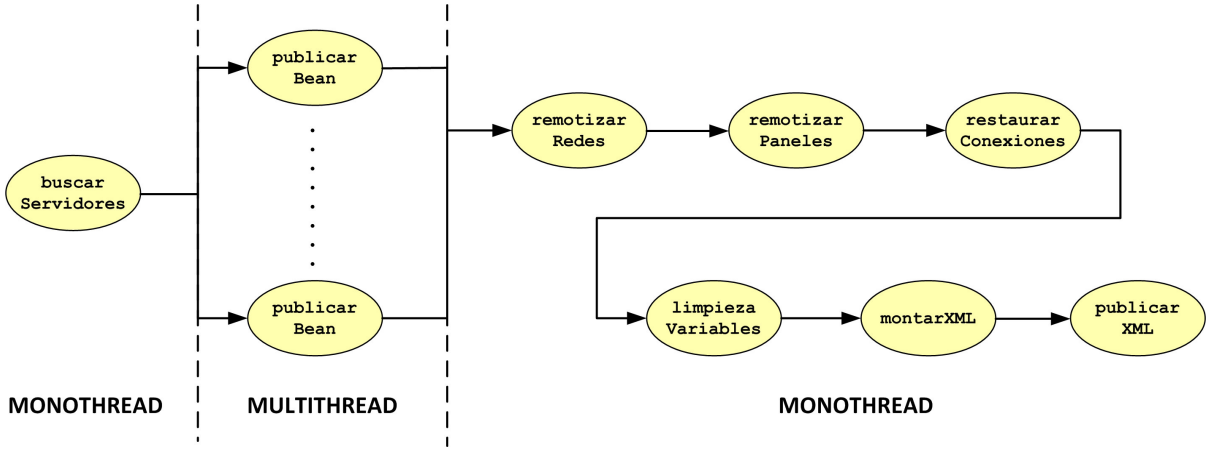

Figura 5.6: Proceso de publicación de una agregación

La primera operación consiste en realizar un sondeo en todo el ámbito de la red para encontrar todos los servidores de *Lookup* en los que se publicará la agregación. Estos servidores serán los pertenecientes a los grupos de *Lookup* seleccionados mediante la interfaz de publicación. Esta tarea se encomienda a *CAEAT Service Manager*.

A continuación se publican individualmente todos los *beans* que pertenecen a la agregación. El proceso de publicación de un *bean* es costoso en términos de tiempo, ya que para cada uno de ellos se debe interactuar con los servidores de *Lookup* a través de la red con tal de registrar en ellos sus objetos públicos. Cuanto más redundada sea la publicación (más servidores LUS se deban "atacar"), más se incrementa este tiempo. Además, para cada *bean* se debe copiar a la carpeta pública del servidor HTTP sus recursos públicos (ficheros *class* y demás recursos asociados), una operación también costosa en tiempo.

Se debe recordar que el proceso de publicación de un *bean* resulta de una estrecha colaboración entre la plataforma CAEAT y el módulo de software autónomo *CAEAT Service Manager*. Dicho proceso fue detallado en el capítulo 4.3.

Si se tiene una agregación formada por pocos componentes, el tiempo de espera es tolerable. Sin embargo, se ha observado que cuando la agregación empieza a crecer (más de 20 – 25 *beans*) los tiempos de espera de publicación llegan a ser molestos y deterioran la experiencia de usuario de la plataforma. Para optimizar el tiempo total de publicación de la agregación, el registro de los *beans* individuales se realiza en un entorno *multithread*, ya que las operaciones que forman el cuello de botella de la publicación son perfectamente realizables de manera concurrente (los servidores de *Lookup* están preparados para atender multitud de peticiones simultáneas).

Se utiliza un pool de N *Threads*, donde N depende del número de *beans* a publicar, siendo en todo caso 10 el número máximo de *beans* asignado a cada *Thread*. El *Thread* principal que está llevando a cabo las tareas de publicación de la agregación espera hasta que los N *threads* lanzados hayan finalizado su trabajo de publicación

A continuación se retoman las operaciones en entorno *monothread*. Se adaptan a la versión remota las instancias de las redes de componentes y los paneles (operaciones necesarias para mantener la coherencia en la capa del Modelo de la aplicación). También se restauran las conexiones del tapiz, que se habrán eliminado en el momento de sustituir las versiones locales de los servicios por sus versiones remotas. Por último, se realiza una limpieza general de diversos objetos de la capa de Modelo de CAEAT que no son necesarios tras completar una publicación.

En este punto la agregación del tapiz ya es remota, pero no se ha publicado su *Proxy* descriptor en los servidores LUS. El último paso a realizar consiste en el montaje del documento XML en base a la agregación del tapiz, la inicialización de los correspondientes objetos `AggregationProxy` y `AggregationServer`, y la publicación del *Proxy* en los mismos servidores de *Lookup* en los cuales se han publicado los sub-componentes.

5.2.4. Publicaciones anidadas

Los argumentos expuestos hasta ahora sugieren que la situación normal de operación a la hora de publicar una agregación será la de disponer de una red de componentes locales y publicarlos todos ellos como una sola agregación. Sin embargo, la herramienta de edición CAEAT se ha adaptado para soportar la anidación de publicaciones.

Lo anterior significa que es posible referenciar agregaciones ya publicadas dentro de la publicación en curso y publicar el resultado como una nueva agregación. Los *beans* locales se publican tal y como se ha descrito, mientras que todo el contenido que ya fuese remoto simplemente se referencia en el documento XML para poder reconstruirlo en el momento de la obtención. La anidación de publicaciones permite realizar operaciones tan potentes como:

- Tomar una agregación de la paleta de servicios remotos, rodearla de componentes locales nuevos y publicar el resultado como una nueva agregación remota.
- Construir y publicar una agregación totalmente nueva sin disponer de ningún componente en el entorno local de CAEAT, únicamente obteniendo y acoplando adecuadamente servicios de la paleta de agregaciones y componentes remotos.

Dado que el documento XML contiene toda la información sobre todas las posibles agregaciones y servicios remotos referenciados, no se producen cambios en el proceso de obtención y reconstrucción de la red.

La siguiente figura muestra una representación visual de la anidación de publicaciones. Sea “Agregación A Publicar” la red que el usuario se dispone a publicar. El usuario ha tomado la

agregación “Agregación Ya Remota” de los servidores de *Lookup*, y la ha rodeado de sus propios *beans* locales para publicar el conjunto como una nueva agregación. Es posible, y sin necesidad de que el usuario tenga conocimiento de ello, que la propia “Agregación Ya Remota” contenga dentro de sí referencias a otras agregaciones, como “Otra Agregación”.

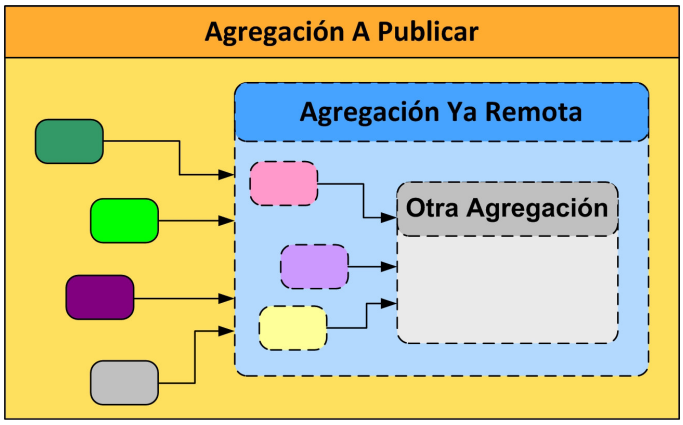

Figura 5.7: Representación visual de una publicación anidada

Cabe destacar que las tres agregaciones mostradas en la figura anterior, pese a encontrarse anidadas en el momento de la publicación, constituyen agregaciones totalmente independientes. Otro usuario podría encontrarse utilizando la agregación “Otra Agregación” de manera concurrente para otras finalidades totalmente distintas. Las operaciones de control de flujo que se pueden llevar a cabo sobre las agregaciones resultan independientes para las tres, como también lo son las propiedades que se presentan en el apartado siguiente.

5.3. PROPIEDADES Y ESTADOS DE LAS AGREGACIONES Y LOS SERVICIOS REMOTOS

Las agregaciones y servicios remotos pueden presentar estados que hasta el momento, en el ámbito de los componentes locales, no se tenían. Además, con la publicación de servicios y agregaciones se introduce el concepto de “propiedad” en el sentido de “posesión”: el usuario poseedor del servicio o agregación gozará de más derechos sobre él que cualquier otro.

Dado que CAEAT pretende ser una plataforma intuitiva y visual, se han utilizado recursos gráficos para transmitir al usuario esta información. Este apartado describe los distintos estados que pueden presentar los servicios y su iconografía asociada en el tapiz de la plataforma.

La primera y más evidente fuente de información es la que diferencia a los componentes locales de los remotos. Estos últimos se representan sobre el tapiz mediante líneas punteadas. Además, el nombre de cada uno de ellos va precedido por el prefijo “[R]”:

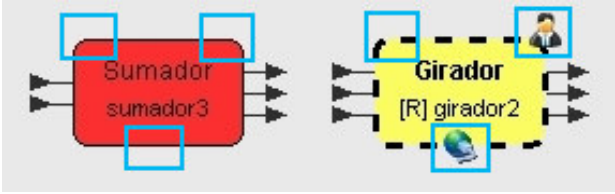

Figura 5.8: Diferenciación entre componentes locales y remotos

Se han reservado asimismo tres espacios alrededor de los componentes (marcados en azul en la ilustración anterior) para transmitir diferentes tipos de información al usuario mediante la inserción de iconos.

5.3.1. Anidamiento de componentes

El espacio superior izquierdo se reserva para indicar al usuario si el componente en cuestión esconde un servicio simple o por el contrario se trata de un componente compuesto, es decir, una agregación que aloja en su interior a más componentes anidados. Esta propiedad es aplicable tanto a los componentes locales como a los remotos.

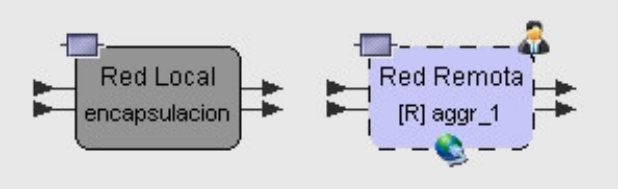

Figura 5.9: Aspecto de agregaciones locales y remotas

Como ya se detalló en el capítulo 2, al realizar doble *clic* sobre un componente simple es posible acceder a la ventana de su *Customizer* para editar sus atributos de manera visual, mientras que al realizarlo sobre un componente compuesto se muestra el esquema interno de éste sobre el tapiz. Para el caso de agregaciones remotas hay excepciones a la regla anterior, que se detallan en el apartado siguiente.

5.3.2. Posesión y opacidad

En el marco de la publicación de servicios se introducen los conceptos de "posesión" y "opacidad". Una mejora realizada durante la elaboración del presente proyecto ha introducido un sencillo mecanismo de gestión de usuarios de la plataforma CAEAT (se explica con más detalle en el anexo D.5). Cada servicio y agregación publicados encapsula un *String* con el nombre del usuario publicador, que pasará a ser el poseedor de esa agregación.

El poseedor de un servicio tiene permiso para realizar las operaciones de control de flujo del servicio que se describen en el capítulo 6, mientras que el resto de usuarios únicamente tiene derecho a utilizarlo. Si el usuario actual de CAEAT es el poseedor de un servicio, se marca con la iconografía que se muestra a continuación:

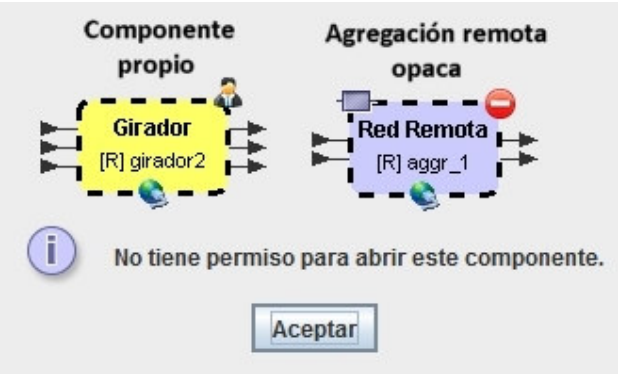

Figura 5.10: Propiedad y opacidad de los servicios de CAEAT

Intrínsecamente ligado al concepto de posesión se encuentra el de opacidad. En el momento de publicar una agregación es posible marcarla como opaca. Esto indica que ningún otro usuario, al margen de su poseedor, podrá abrir la agregación mediante un doble *clic* y navegar y editar sus componentes internos. Ésta es la opción por defecto, ya que implementa a la perfección el concepto de "caja negra" que se ha expuesto hasta ahora. Si una agregación es opaca, se muestra un icono de prohibición y no se permite a los usuarios abrirla en el momento en que lo intentan.

Existe una tercera posible combinación: el usuario puede no tener la posesión de una agregación, pero en cambio ser ésta "transparente" (no haber sido publicada como opaca). En este caso, no se muestra ningún icono y el usuario tiene permiso para navegar por los componentes internos de la agregación. Sin embargo, sigue sin tener permiso para ejecutar las acciones de control de flujo del servicio reservadas únicamente al poseedor del mismo (ver capítulo 6).

5.3.3. Estado de publicación y supervivencia

El último de los espacios de inserción de iconografía, el inferior central, es aplicable únicamente a los servicios remotos y se ha reservado para la información relativa al estado de publicación y supervivencia del servicio. Los posibles iconos que se muestran en este espacio pueden observarse en la figura siguiente:

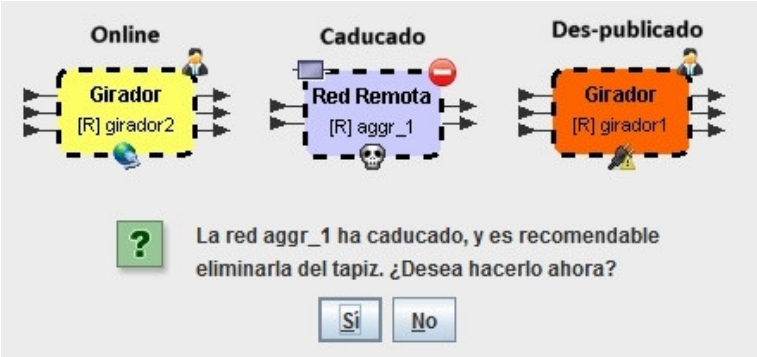

Figura 5.11: Posibles estados de publicación y supervivencia de los servicios de CAEAT

Los tres estados de publicación y supervivencia en los que puede encontrarse un servicio son los siguientes:

Online: el servicio está operando con normalidad y el servidor responde correctamente a las llamadas remotas que se le realizan desde CAEAT.

Caducado: todos los servicios implementan un método llamado `isAlive()` con la finalidad de sondear al servidor para comprobar si se encuentra operativo. Se ha implementado en CAEAT un mecanismo que sondea a todos los servicios insertados en el tapiz cada cierto tiempo, para cerciorarse de que todos se encuentran online.

Si se suceden algunas llamadas a `isAlive()` infructuosas, se da por hecho que el servidor ha caído por alguna razón (detención manual del servicio en la máquina en la que reside, fallo de hardware o de la red, etc.), y su componente asociado se marca como caducado en el tapiz de CAEAT. En este estado, la única acción realizable sobre ese componente es su eliminación, que es propuesta al usuario si éste intenta hacer uso del componente caducado.

Una agregación remota se considera caducada si alguno de sus componentes internos ha caducado (es suficiente la expiración de un sub-servicio para que la funcionalidad de la agregación "como un todo" quede seriamente comprometida).

Des-publicado: una de las operaciones de control de flujo de los servicios descritas en el capítulo 6 es la des-publicación. Bajo este estado, el servicio se mantiene activo, pero sus objetos públicos son retirados de los servidores de *Lookup*.

Existe un cuarto icono que puede mostrarse en este espacio de los componentes: una advertencia para indicar que una agregación remota necesita de actuación humana tras haber sido modificada en caliente (ver apartado 5.6).

5.4. BÚSQUEDA, OBTENCIÓN Y UTILIZACIÓN DE REDES REMOTAS Y SERVICIOS

Hasta el momento se ha detallado el proceso de publicación y mantenimiento de agregaciones y servicios, es decir, se ha centrado el discurso en el punto de vista del servidor (entidad que exporta y ofrece el servicio al resto de clientes de la red). El presente apartado realiza una breve descripción de la entidad opuesta: el cliente que busca, obtiene y utiliza dichos servicios.

5.4.1. Paleta de servicios remotos

Cuando un cliente de CAEAT desea hacer uso de un servicio, realiza una búsqueda por la red mediante el uso de las herramientas de *Apache River* que permiten realizar un proceso de *discovery*, protocolo explicado en el capítulo 1. Los resultados de dicha búsqueda serán los *proxies* de los servicios que se encuentran registrados en los servidores de *Lookup*.

Para facilitar enormemente la inserción de servicios en el tapiz, y para mejorar la experiencia de usuario, se ha elaborado una segunda paleta de servicios en la interfaz de la plataforma. Esta nueva paleta comparte espacio con la paleta de *beans* locales mediante una conmutación por pestañas. Como paso previo a una búsqueda de servicios, primero se determina el número de grupos de *Lookup* disponibles en la red mediante un sondeo rápido, añadiendo a la paleta tantos nodos como grupos distintos se hayan encontrado. A continuación se realiza el *discovery* y se insertan progresivamente los servicios en la paleta a medida que se encuentran, convenientemente clasificados según su grupo de registro de procedencia.

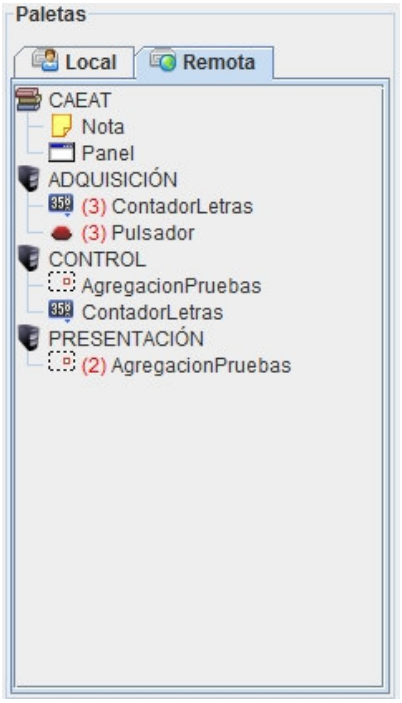

Figura 5.12: Paleta de servicios remotos de la interfaz de usuario de CAEAT tras una búsqueda

Si se reciben *proxies* duplicados, significa que la publicación de ese servicio está redundada (es decir, que los objetos del servicio se han registrado en más de un servidor de *Lookup*). En estos casos se añade al nombre del servicio un número que da al usuario una idea del nivel de redundancia de la publicación. Se debe recordar que para el caso de agregaciones remotas únicamente se mostrará el *proxy* de la agregación, no de sus sub-servicios. La conveniencia de la elección de los nombres de los grupos de *Lookup* que se muestran en la figura anterior se explica en el apartado siguiente.

Para estar convenientemente al día pero no sobrecargar el software y la red con búsquedas constantes de servicios, la paleta remota se borra y rellena de manera periódica cada N minutos a

partir de una nueva búsqueda general de servicios. N es un número seleccionable por el usuario mediante el menú de preferencias implementado en CAEAT. El fondo de la paleta remota parpadea cuando se está llevando a cabo una búsqueda para informar al usuario de que aún pueden encontrarse y listarse nuevos resultados.

5.4.2. Documento XML descriptor de una agregación

Las agregaciones remotas vienen definidas por un documento XML que se obtiene a través del *proxy* de su servicio, y que permite reconstruir la red en el lado del cliente tal y como se publicó en el lado del servidor. A continuación se muestra el formato de un documento XML descriptor de una agregación remota sencilla, formada únicamente por dos componentes conectados entre sí:

```
<Esquema_Remoto Version="1" NombreRed="Agregación 2 componentes" Propietario="sergio"
GruposRegistro="ADQUISICIÓN;PRESENTACIÓN" RandomID="-5377098811538986684">
  <Componentes>
    <Componente_Remoto NombreInstancia="[R] pulsador1" Color="-6684775"
NombreClase="Pulsador" Propietario="sergio" ServicioAutonomo="NO">
      <X>300</X>
      <Y>339</Y>
      <RandomID>-18043320286057555</RandomID>
      <IDLow>-5013782459681752889</IDLow>
      <IDHigh>124912433372284534</IDHigh>
    </Componente_Remoto>
    <Componente_Remoto NombreInstancia="[R] contadorletras2" Color="-154"
NombreClase="ContadorLetras" Propietario="sergio" ServicioAutonomo="NO">
      <X>499</X>
      <Y>464</Y>
      <RandomID>7059785573302944029</RandomID>
      <IDLow>-8635974774005066634</IDLow>
      <IDHigh>-4315038908184509090</IDHigh>
    </Componente_Remoto>
  </Componentes>
  <Conexiones>
    <Conexion Color="-16776961">
      <Origen Patilla="estado" Componente="[R] pulsador1" RandomID="-18043320286057555" />
      <Destino Patilla="cuenta" Componente="[R] contadorletras2"
RandomID="7059785573302944029" />
    </Conexion>
  </Conexiones>
  <Patillas>
    <Patilla ComponentePadre="[R] contadorletras2" NombrePropiedad="letras"
NombrePatilla="letras" RandomID="7059785573302944029" />
    <Patilla ComponentePadre="[R] contadorletras2" NombrePropiedad="mensaje"
NombrePatilla="mensaje" RandomID="7059785573302944029" />
    <Patilla ComponentePadre="[R] contadorletras2" NombrePropiedad="cuenta"
NombrePatilla="cuenta" RandomID="7059785573302944029" />
  </Patillas>
  <Opacidad>SI</Opacidad>
  <Paneles />
  <Notas />
</Esquema_Remoto>
```

Código 5.1: Documento XML descriptor de una agregación remota

El formato escogido es muy similar al que ya se utilizaba para el guardado de las redes en el entorno local (ficheros *aec*). Las novedades y diferencias se detallan a continuación:

`GruposRegistro`: cada agregación es marcada con un listado que contiene los nombres de los grupos de registro en los que se ha registrado la agregación (y consecuentemente, todos sus sub-servicios), con tal de facilitar la búsqueda y reconstrucción en recepción.

`ServiceID`: cada sub-servicio es marcado con una representación en cadena de caracteres de su `ServiceID`. Este parámetro es necesario en recepción para poder obtener el servicio y reconstruir la red. Dado que el `ServiceID` es un identificador de 128 bits, *Jini* ofrece facilidades para manejarlo como un par de números de tipo *long* (64 bits cada uno), que se representan por separado en el XML (`IDHigh` e `IDLow`).

`Propietario`: al reconstruir la agregación es necesario marcar cada componente con el nombre del usuario poseedor de cada uno de ellos, para determinar los privilegios sobre el servicio que tendrá el usuario actual de CAEAT. Se incluye dicho nombre en cada uno de los servicios especificados en el XML.

`Opacidad`: especifica si la agregación se ha publicado como opaca.

`ServicioAutonomo`: especifica si ese componente / agregación representa un servicio publicado independientemente o si por el contrario es un sub-servicio cuyo flujo de vida y propiedades están ligados a los de una agregación padre.

`RandomID`: número aleatorio que se utiliza en la lógica de edición de redes remotas en caliente (capítulo 5.6), para poder identificar eficazmente diferentes instancias de un mismo servicio sin tener que atender únicamente al `ServiceID`.

El resto de parámetros siguen la misma estructura y formatos que en la versión local del documento XML: posición de los componentes, colores, conexiones, posibles notas que pudiese contener el esquema, paneles contenedores de elementos gráficos, etc.

5.4.3. Reconstrucción de la agregación en recepción

La reconstrucción de la agregación tiene lugar automáticamente cuando el usuario decide tomar un servicio desde la paleta remota e insertarlo en el tapiz. Los pasos que se llevan a cabo para conseguir la reconstrucción de la agregación son los siguientes:

1. Obtención del documento XML descriptor de la agregación mediante el *proxy*, por medio de una llamada remota al servidor que aloja dicho documento.
2. Determinación de los grupos de registro a consultar. Realización de un *discovery* rápido en toda la red para obtener los objetos `ServiceRegistrar` que permitan a CAEAT comunicarse con todos los servidores LUS pertenecientes a dichos grupos de registro.
3. Para cada componente remoto descrito por el documento XML, se realiza una búsqueda de su *proxy* dentro de los servidores Lookup obtenidos en el paso anterior. Dado que las búsquedas se pueden parametrizar de distintas maneras, se busca por medio del `ServiceID`, dato incluido en el documento XML. Para cada *proxy* obtenido, se obtiene también su objeto *wrapper* y se genera un componente listo para ser insertado en el tapiz de CAEAT. Si el documento XML contiene la descripción de más redes remotas anidadas, se buscan sus componentes recursivamente de idéntica forma.
4. Se reconstruyen los demás aspectos de la red tal y como se realizaba en el caso local: encapsulaciones de componentes, colores, posicionamiento, conexiones, etc.

Una barra de progreso informa al usuario de los avances en la obtención de los sub-servicios de la agregación. A diferencia de la publicación, el proceso de obtención se realiza completamente bajo una disciplina *monothread*. Se ha comprobado que no es necesario el uso de varios *threads* ya que la obtención de servicios es mucho más rápida que su publicación por las siguientes razones:

- No se debe realizar la copia al sistema de archivos local de ningún fichero, operación muy costosa en términos de tiempo si se hace repetidamente. Se debe recordar que los archivos *class* de las clases cuyos objetos se obtienen de manera dinámica se cargan desde la ubicación remota del servidor, pero nunca se descargan al sistema de ficheros local.
- No se debe contactar con más de un servidor de *Lookup* por servicio obtenido. Pese a que el *proxy* del servicio esté redundado, éste se obtiene del primero de los servidores LUS que contesten a la petición.

5.5. GRUPOS DE REGISTRO EN EL ENTORNO CAEAT

En este apartado se detalla por qué se le ha dado tanta importancia a la característica de los grupos de *Lookup* ofrecida por *Jini / Apache River* en el contexto de CAEAT y *SeNetComponents*. También se explican las motivaciones del uso de los tres grupos de *Lookup* que se han podido observar en las figuras adjuntas del presente capítulo: "ADQUISICIÓN", "CONTROL" y "PRESENTACIÓN".

En el capítulo 2 se explicó que la finalidad última de una plataforma de despliegue y publicación de componentes software como *SeNetComponents* es servir como sistema de control, adquisición y monitorización de datos obtenidos por dispositivos hardware ubicados en campo. En este contexto, es útil y habitual dividir el procesado en tres capas claramente diferenciadas:

- **Capa de adquisición**: es la responsable de la interacción con los dispositivos hardware que adquieren las magnitudes físicas. En el contexto CAEAT / *SeNetComponents*, esta capa ejecutaría agregaciones que encapsularían los *drivers*, protocolos y la lógica de interacción con la electrónica a monitorizar (PLC's, PAC's, etc.).
- **Capa de control**: es la responsable de hacer de "puente" entre los usuarios finales y los dispositivos de campo. En el sentido dispositivos - usuarios, se toman los datos necesarios de la capa de adquisición y se procesan adecuadamente con tal de presentarlos al usuario. En el sentido contrario, se propagan las órdenes e *inputs* de los usuarios hacia los dispositivos. Adicionalmente se pueden llevar a cabo acciones sobre los datos adquiridos, como su almacenamiento en una base de datos. En el contexto de CAEAT / *SeNetComponents*, las agregaciones pertenecientes a esta capa encapsularían a las de adquisición, y dejarían sus *outputs* expuestos a la siguiente y última capa.
- **Capa de presentación**: es la responsable de obtener los *outputs* finales generados por las otras capas y presentarlos al usuario mediante la interfaz hombre – máquina adecuada. En el contexto de CAEAT / *SeNetComponents*, las agregaciones de presentación encapsularían a las de control y las rodearían de los elementos gráficos y de entrada / salida necesarios para proporcionar una interfaz de control adecuada.

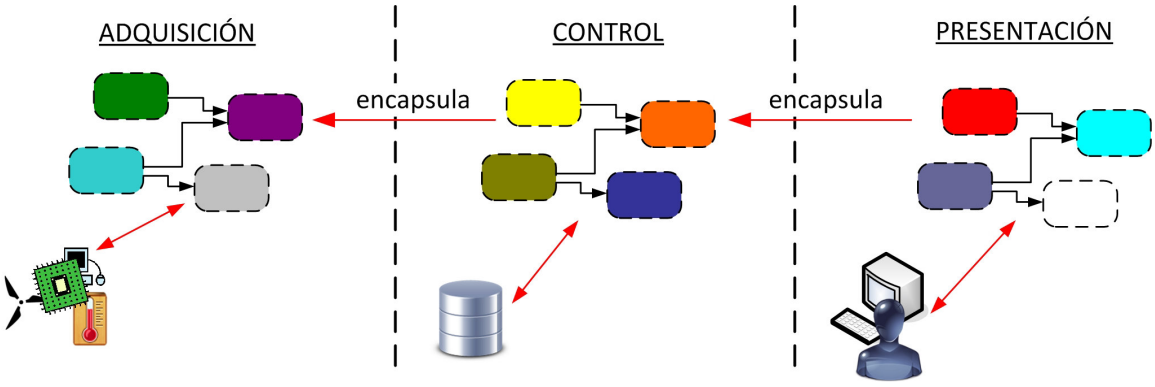

Figura 5.13: Arquitectura de procesado tricapa aplicable a la plataforma *SeNetComponents*

Bajo una arquitectura tricapa será muy importante mantener aisladas entre sí agregaciones pertenecientes a capas diferentes. En un entorno de edición como CAEAT se podrán aplicar limitaciones a los usuarios, de manera que aquéllos encargados de trabajar con una capa no puedan ver los servicios que están corriendo en las otras. La posibilidad de agrupar los servidores de *Lookup* en diferentes grupos de registro es una característica que encaja a la perfección con esta filosofía de diseño.

Con esta arquitectura en mente, el presente proyecto se ha desarrollado permanentemente bajo una configuración de 3 grupos de registro. Sin embargo, como se puede observar en las interfaces de usuario implementadas, no se ha cerrado ninguna puerta a que en el futuro puedan aparecer nuevos grupos.

5.6. MODO DE EDICIÓN EN CALIENTE DE AGREGACIONES REMOTAS

En la plataforma CAEAT, la situación más habitual de publicación será la opacidad; es decir, como norma general únicamente el poseedor de una red ya publicada podrá navegar por los componentes internos de la misma y realizar modificaciones. Sin embargo, no será del todo extraño que más de un usuario, de forma concurrente, tenga acceso al interior de una misma agregación desde máquinas distintas de la red. Esta situación se puede dar si por ejemplo:

- El poseedor de la red no la ha marcado como opaca ya que considera que debe ser editable por todos los usuarios.
- Se amplía el mecanismo de gestión de usuarios de manera que se permita la posesión compartida da agregaciones o se crea la figura del super-usuario que tiene acceso en modo edición a todas las agregaciones de la red.

En previsión de esta situación, se ha implementado un método de actualización en caliente de las modificaciones sufridas por agregaciones remotas ya publicadas.

5.6.1. Objetivos y propósitos

En el contexto de la plataforma de edición CAEAT, se ha querido dotar a las agregaciones remotas del mismo comportamiento que cualquier otro servicio remoto simple. Los servicios simples contienen propiedades, generalmente tipos de datos primitivos del lenguaje de programación *Java*, que pueden ser modificados por los usuarios o por otros servicios, y que se mantienen actualizados y en sincronía entre todos los clientes del servicio por medio del mecanismo de lanzamiento de eventos ampliamente detallado en capítulos anteriores.

Las agregaciones remotas pueden ser vistas como un meta-servicio remoto cuya única propiedad radica en su estructura interna y los sub-servicios que la componen, información que se encapsula en el documento XML que reside en el servidor y que está convenientemente protegido contra acceso concurrente como ya se detalló que se hacía con las propiedades de los servicios en el capítulo 3.5. Dado que el *proxy* de la agregación ofrece métodos *get* y *set* para consultar y establecer el XML, los clientes poseen la capacidad de establecer nuevas versiones de dicho documento.

Bajo esta visión, se ha diseñado un mecanismo de edición en caliente de las agregaciones remotas que permite la actualización de la estructura de la red y la consiguiente notificación de los cambios a todos los clientes de dicha agregación; consiguiendo así la sincronización instantánea de las modificaciones realizadas sobre la agregación.

Se consigue de esta manera dotar a las agregaciones de un carácter “vivo”, ya que es posible realizar sobre ellas todo tipo de modificaciones en tiempo de ejecución (sin necesidad de detenerlas o des-publicarlas) y que éstas modificaciones sean “disparadas” al instante al resto de entidades que puedan estar haciendo uso de la agregación.

5.6.2. Implementación

Se ha implementado el “modo de edición en caliente” en la plataforma CAEAT. Dicho modo se activa cuando un usuario con privilegios sobre una red del tapiz decide abrirla (realizando doble *clic* sobre ella) y navegarla para realizar modificaciones sobre sus componentes internos. En el modo de edición en caliente, la superficie del tapiz de CAEAT se oscurece para indicar al usuario la entrada a dicho modo:

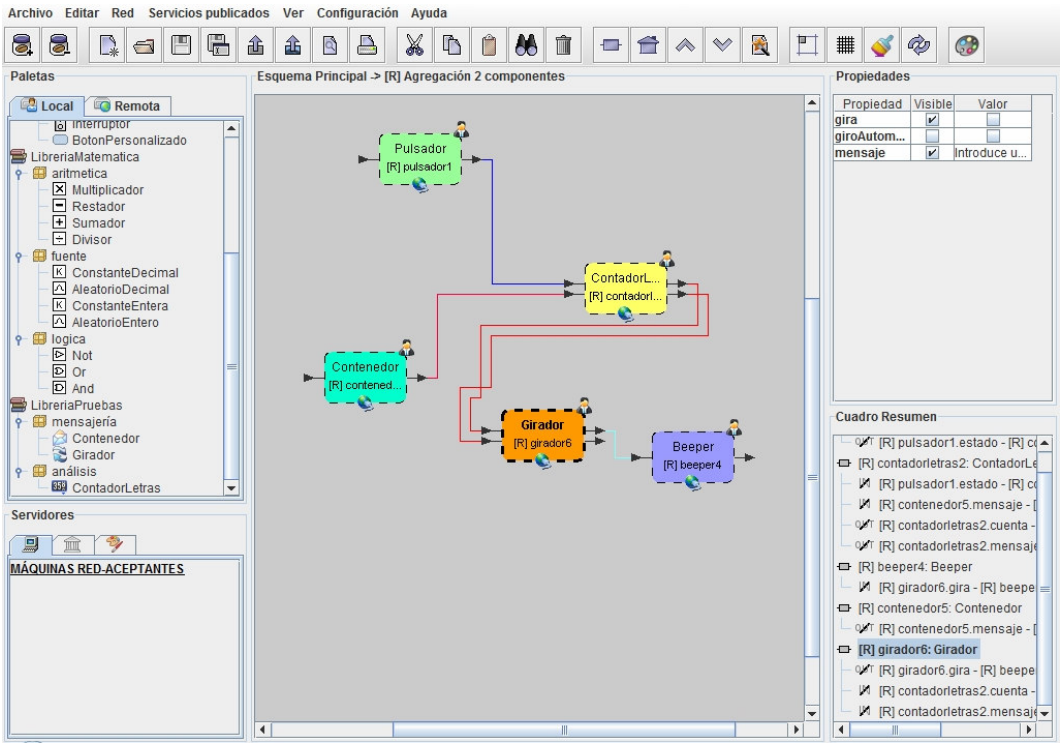

Figura 5.14: Interfaz de CAEAT en el modo de edición en caliente de agregaciones remotas

Cuando en el modo de edición en caliente se produce una modificación que debe ser inmediatamente actualizada, se procede a la construcción del que a partir de ahora será el nuevo documento XML descriptor de la agregación. Dicho documento se deposita en el servidor a través del *proxy* de la agregación. El servidor es entonces el responsable de lanzar un evento remoto a todos los clientes que están haciendo uso de esa agregación para informarles del cambio que ha tenido lugar.

En el capítulo 3.5 se detalló el mecanismo de lanzamiento de `RemoteEvent`'s a los clientes para notificarlos de diferentes circunstancias. Uno de los posibles tipos de eventos que CAEAT está preparado para tratar es el de notificación de cambio estructural en una agregación del tapiz. Como ya se explicó en el mencionado capítulo, el objeto `RemoteEvent` del entorno *Jini* ofrece al programador una gran libertad en cuanto a estructura y contenido. Es por ello que en este tipo de eventos se encapsula el nuevo documento XML de la agregación y un campo de información sobre el cambio que ha tenido lugar.

Bajo la recepción de un evento de estas características, CAEAT determina en primera instancia el tipo de cambio que ha tenido lugar para después, mediante análisis del nuevo documento XML, llevar a cabo las actualizaciones pertinentes en función de dicho cambio. Todas las modificaciones que se pueden llevar a cabo sobre una agregación en el tapiz de CAEAT se resumen en 7 cambios atómicos que se detallan en el apartado siguiente.

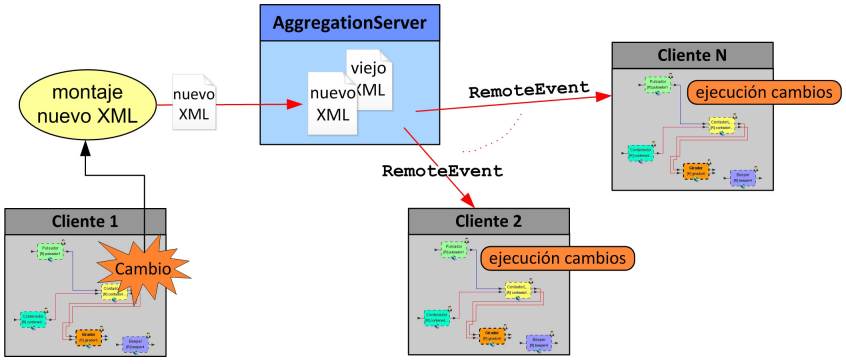

Figura 5.15: Lógica de actualización automática en caliente de las agregaciones remotas

5.6.3. Modificaciones atómicas de las agregaciones en caliente

Las 7 modificaciones atómicas que se pueden llevar a cabo sobre una agregación remota en el modo de edición en caliente son las siguientes:

5.6.3.1. Inserción de un nuevo servicio

Si se trata de un servicio provinente de la paleta remota (servicio ya remoto), se construye el nuevo XML y se disparan los cambios. Si por el contrario se trata de un *bean* local, éste se publica automáticamente como paso previo al montaje del nuevo XML. Los datos de publicación del nuevo *bean* a publicar (como por ejemplo los grupos de *Lookup* que debe acometer) serán los mismos que los de la agregación que se está editando. Si por algún motivo el *bean* insertado no fuese publicable, se informa al usuario y no se lleva a cabo la operación.

En recepción, se realiza una búsqueda en la red del nuevo servicio mediante su `ServiceID`. Se inserta convenientemente en el tapiz respetando los datos (nombre, posición, color) que marca la nueva versión del XML recién recibida.

5.6.3.2. Eliminación de un servicio

El montaje del nuevo XML es inmediato tras la eliminación de un servicio. En recepción, se localiza el servicio objetivo mediante el parámetro `RandomID` presentado en un apartado anterior. Este dato es necesario puesto que confiando únicamente en el `ServiceID` se correría el riesgo de eliminar el componente equivocado en caso de que el cliente tuviese múltiples instancias de ese servicio en el tapiz.

CAEAT permite la multi-selección de varios componentes al mismo tiempo y su eliminación simultánea: con el propósito de atomizar los eventos al máximo, se generan y lanzan secuencialmente tantos eventos como componentes se estén eliminando al mismo tiempo.

5.6.3.3. Creación de una conexión

Tras la creación de una nueva conexión, se crea y "dispara" un nuevo documento XML. En recepción se extrae información sobre los componentes origen y destino de la conexión, así como las patillas de dichos componentes que están conectados a través de ella. Con esta información, se recrea la conexión, respetando también aspectos no-críticos como su color.

5.6.3.4. Eliminación de una conexión

Tras la eliminación de una conexión, la creación de un nuevo XML es automática e inmediata. En recepción, el parámetro `RandomID` y la información sobre los componentes y patillas origen y destino ayudan a identificar unívocamente la conexión que se debe eliminar.

5.6.3.5. Cambio de patillas externas

Este cambio tiene lugar cuando el poseedor de la agregación decide cambiar los atributos que serán visibles desde el exterior. Esta modificación se debe tratar como un caso especial, ya que no consiste únicamente en un cambio en las patillas externas de un componente: el poseedor de la red, en el fondo, está redefiniendo su interfaz pública con el resto de componentes de las distintas agregaciones. El cambio de patillas externas se puede efectuar desde el menú contextual del tapiz de

CAEAT, desde el cual se accede a una tabla de selección de patillas idéntica a la que se muestra en la interfaz de publicación (ver figura 5.2).

El montaje y "lanzamiento" de un nuevo documento XML es sencillo, y las particularidades surgen en recepción. La estructura interna de la agregación no cambia, pero sí la "caja negra" que la representa en el tapiz y que, posiblemente, la interconecta con otros componentes. La información contenida en el nuevo XML es suficiente para construir la nueva "caja negra", pero no así para reconstruir las conexiones que pudiese tener. Un cambio de patillas externas puede suponer también un cambio en sus nombres; por lo que la figura del programador no tiene potestad para decidir restaurar algunas de las conexiones antiguas en base a ese u otros datos.

La solución adoptada pasa por la desconexión total de la nueva "caja negra" de todos los componentes externos a los que pudiese estar conectada previa recepción del evento de cambio de patillas. Se informa al usuario de que el componente puede necesitar de una intervención manual (restauración de las conexiones) mediante la inserción de un icono de advertencia parpadeante en el lugar en el que habitualmente se muestra el icono que indica el estado de vida del servicio.

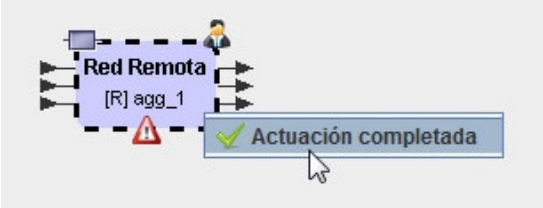

Figura 5.16: Agregación remota marcada con la necesidad de una intervención manual

El componente resta funcionando con normalidad, pero no se permite ninguna acción especial sobre el servicio que representa (detención, cambio de nombre, nuevo cambio de patillas, etc.) hasta que el usuario marca que la intervención ha sido completada. Esta acción se realiza mediante el menú contextual del componente (ver figura anterior).

5.6.3.6. Modificación de elementos mostrados en los paneles

La eliminación o adición de elementos gráficos de los paneles de la agregación se considera un cambio crítico, pues constituye una modificación de la interfaz hombre-máquina de dicha agregación. Cuando los elementos contenidos en un panel son modificados, se reconstruye y se lanza un nuevo documento XML. En recepción, se localiza el panel afectado y se vacía temporalmente de elementos, para a continuación rellenarlo con los nuevos elementos que especifica el XML, que se deberán buscar en el esquema. Al finalizar se realiza un repintado general del panel y todos los componentes incluidos en él.

Puede sorprender el hecho de que esto implique que todos los clientes que en ese momento estuviesen haciendo uso de la interfaz gráfica de la agregación la vean abruptamente modificada, pudiendo aparecer elementos nuevos y desapareciendo los obsoletos. Aunque sorprendente, éste es el comportamiento que se busca en una herramienta de edición concurrente como es CAEAT.

5.6.3.7. Modificaciones no-estructurales

Existen ciertas operaciones que CAEAT permite realizar sobre los esquemas que, en el contexto del modo de edición en caliente de agregaciones remotas, no se consideran modificaciones críticas. Se trata de operaciones que no afectan al comportamiento de la agregación a nivel funcional, y que únicamente acarrean consecuencias visuales y "decorativas". Dichos cambios son los siguientes:

- Cambio en el nombre de instancia de los componentes del tapiz

- Cambio en el posicionamiento y distribución de los componentes sobre el tapiz
- Cambios en los colores de los componentes y las conexiones
- Inserción de notas recordatorias (*post-it's*) sobre el esquema

Dado que estos cambios no comprometen la integridad de la red, no se disparan automáticamente al ser realizados. En lugar de eso, se ofrece al usuario la posibilidad de dispararlos bajo demanda mediante un botón situado en uno de los menús desplegables. En caso de recibirse un evento de este tipo, la entidad receptora realiza un recorrido por los componentes de la agregación y compara sus nombres, posiciones y colores con los del nuevo XML recibido, y actualiza los necesarios en consecuencia. Del mismo modo, se itera sobre todas las notas *post-it* y se eliminan o añaden las que sean necesarias. Al finalizar el proceso se realiza un repintado general del esquema.

5.7. AGREGACIONES REMOTAS Y ARCHIVOS AUTOEJECUTABLES

En el capítulo 2.8 se expuso la metodología mediante la cual es posible exportar una agregación generada mediante la herramienta CAEAT a un archivo *jar* autoejecutable. La ejecución de dicho archivo desplegaba la red en un entorno ajeno a CAEAT, por lo que no se permitía la edición ni la visualización de sus componentes, únicamente su utilización mediante la interfaz gráfica creada para ella desde CAEAT (en caso de que la agregación proporcionase dicha interfaz).

Uno de los objetivos del presente proyecto es la adaptación de esta lógica de ejecución de las agregaciones al entorno distribuido. Para ello, se han incluido en el archivo autoejecutable las clases de CAEAT necesarias para, además de permitir la ejecución en local de la agregación, conseguir también:

- Su exportación como servicio y publicación según lo expuesto en este capítulo
- Su lanzamiento hacia una máquina red-aceptante (el capítulo 7 describe este concepto)

En el momento de la generación de un archivo *jar* autoejecutable, se comprueba si todos los componentes contenidos en él son compatibles con la publicación. Si existen componentes no compatibles, el autoejecutable únicamente podrá ser desplegado en el entorno local, volviendo así al caso particular que ya se había implementado antes del inicio del presente proyecto.

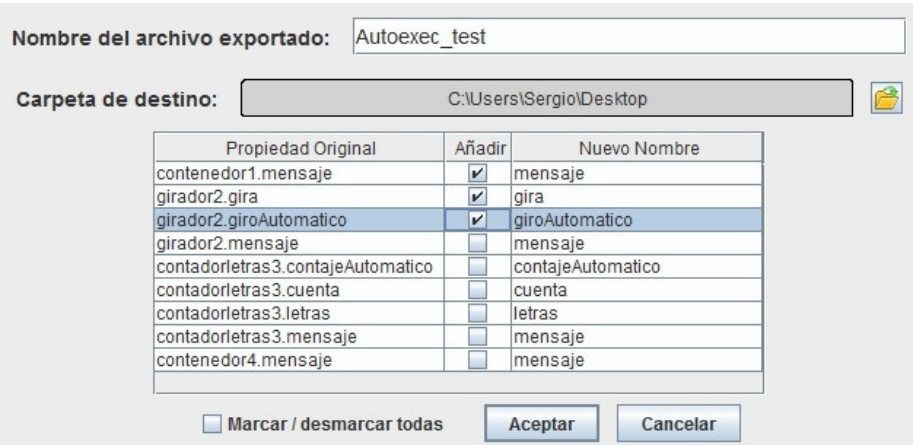

Figura 5.17: Interfaz de exportación de una agregación hacia un archivo autoejecutable

En caso contrario (todos los componentes pueden ser publicados), el usuario generador del autoejecutable debe definir su interfaz externa. Este proceso tiene la misma finalidad que en el caso de la publicación desde el entorno CAEAT: la definición de las propiedades que serán visibles y editables desde el exterior de la agregación en forma de patillas. Nótese que la responsabilidad de la selección de dicha interfaz no debe recaer sobre el usuario que publica la agregación contenida en el

archivo *jar* (puesto que no conoce su estructura interna), sino sobre aquél que la crea desde CAEAT. La información del mapa de propiedades externas es almacenada en el archivo *aec* junto con los componentes que forman la agregación.

Desde el punto de vista del usuario que despliega el autoejecutable, es posible seleccionar entre su ejecución en local, su publicación desde la máquina local o su lanzamiento a una máquina red-aceptante. En el caso de la publicación, se ofrece al usuario una interfaz de selección de datos idéntica a la presentada en el capítulo 5.2, a excepción de la tabla de selección de propiedades externas (este dato ya ha sido establecido por el creador del autoejecutable y resulta inamovible). En el caso de desear lanzar la red, se realiza una búsqueda de máquinas red-aceptantes por toda la federación y se muestran éstas en forma de tabla, pudiendo elegir una hacia la cual lanzar la agregación (véase el capítulo 7 para más información sobre este proceso).

Las agregaciones publicadas desde un entorno autónomo pueden ser obtenidas y utilizadas de idéntica forma que las publicadas desde la plataforma CAEAT, pudiendo interconectarlas con cualquier otro componente, ya sea servicio remoto o *bean* local. Sin embargo, se ha impuesto como criterio de diseño que las agregaciones remotas provinentes de publicaciones llevadas a cabo desde entornos autónomos a CAEAT no sean propiedad de ningún usuario en concreto, siendo por lo tanto opacas y no editables una vez completada su publicación. La ampliación del sistema de gestión de usuarios introducido durante la elaboración del presente proyecto deberá añadir complejidad y nuevas funcionalidades a la gestión del ciclo de vida de las agregaciones autónomas (véanse las líneas de trabajo abiertas en el capítulo 9.2).

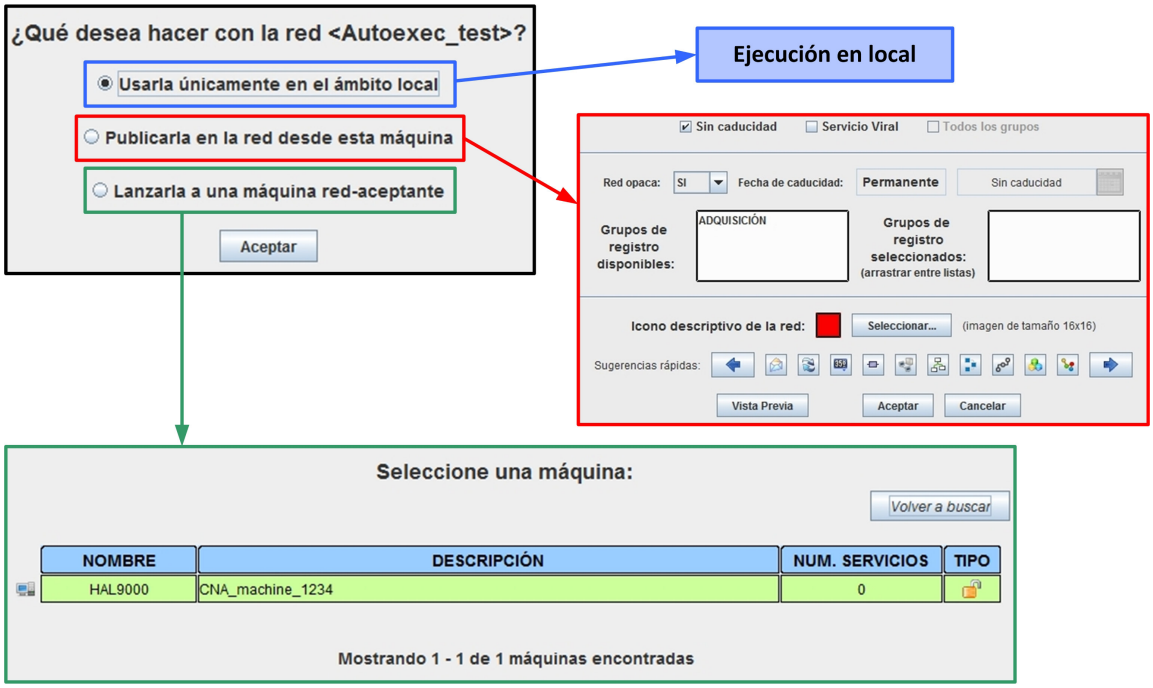

Figura 5.18: Lógica de ejecución, publicación o lanzamiento de una agregación autoejecutable

6. CONTROL DE FLUJO DE LOS SERVICIOS

Hasta este momento se han expuesto los procedimientos empleados para exportar, publicar e inicializar servicios *Jini / Apache River* en el entorno formado por *SeNetComponents* y la herramienta de edición CAEAT. Sin embargo, una vez iniciados los servicios, resulta útil proporcionar herramientas de actuación sobre su ciclo de vida. Por lo general, un servicio restará activo hasta la llegada de su fecha de caducidad, pero es muy probable que con el transcurso del tiempo se decidan realizar cambios que afecten al ciclo de vida natural del servicio.

Se han implementado cuatro procedimientos especiales de control del flujo de la vida de los servicios: su detención, el cambio en su fecha de caducidad, su reinicio y su des-publicación. Este capítulo aborda los mecanismos diseñados para la implementación de dichas funcionalidades. Se expone también el concepto de "servicio viral", que afecta sobremanera al ciclo de vida de las publicaciones realizadas.

Las cuatro operaciones especiales sobre el ciclo de vida de los servicios podrán ser ejecutadas únicamente por los propietarios de los mismos. Pese a que una agregación sea no opaca y por lo tanto editable por cualquier usuario, únicamente el propietario podrá llevar a cabo las operaciones especiales. Estas operaciones pueden realizarse desde el menú contextual de los componentes del tapiz si se es el propietario del servicio, tal y como muestra la siguiente figura:

Figura 6.1: Operaciones especiales de control de flujo de los servicios accesibles desde el tapiz

Las operaciones especiales de control del flujo de vida de los servicios, por lo general, pueden ejecutarse remotamente. Esto significa que el propietario del servicio no necesita encontrarse trabajando en la máquina en la cual el servicio está siendo ejecutado, sino que puede solicitar su ejecución de manera remota a través del *proxy* de los servicios. Existe alguna excepción a esta regla, que se detalla en la explicación de la operación en cuestión.

6.1. DETENCIÓN DE UN SERVICIO

La primera y más lógica de las operaciones especiales a ejecutar sobre un servicio es su detención. Pese a que los servicios se inicializan con fecha de caducidad, pueden darse diversas circunstancias que provoquen su obsolescencia prematura (por ejemplo, el lanzamiento de una nueva versión de una librería de componentes que podría introducir mejoras en ese servicio y llevar por consiguiente a una actualización). Es necesario, pues, un mecanismo de detención manual de servicios que cumpla los siguientes requisitos:

- Siguiendo la filosofía de arquitectura orientada a servicios, la detención de un servicio debe ser transparente a la máquina desde la cual se está deteniendo. Si el servicio fue exportado en la misma máquina desde la cual se está deteniendo, la detención se realizará en local; si se realiza desde otra máquina, se ejecutará mediante una llamada remota.

- Los servidores de *Lookup* deben quedar limpios de todo objeto relacionado con el servicio detenido. Se deben cancelar explícitamente los *leases* o concesiones de todos los objetos *Proxy* y *Wrapper* que formaban parte tanto del servicio como de sus sub-servicios (en caso de tratarse de una agregación de servicios).
- De manera similar, todos los objetos *Server* que fueron exportados en la máquina servidora para atender las llamadas remotas de los clientes serán des-exportados. *CAEAT Service Manager* debe realizar una limpieza de todas las referencias a objetos relacionados con el servicio para garantizar que no quedan residualmente en la memoria de la JVM.
- La detención de los servicios debe de ser "limpia" de cara a los clientes que estuviesen haciendo uso de él. En el momento de la detención se enviará una notificación de caducidad a todos los clientes para advertirles de que el servicio deja de estar disponible.
- Si una agregación de servicios remotos contiene otros servicios y agregaciones autónomos (que ya habían sido publicados con anterioridad a la agregación que se está deteniendo), éstos no se detienen. Puede darse el caso que estos servicios autónomos internos no pertenezcan al mismo usuario que la agregación que se está deteniendo, e incluso que hayan sido encapsulados por terceros usuarios en otras agregaciones distintas y estén siendo utilizados en otros puntos de la federación de servicios.

6.1.1. Implementación

Para poder dar soporte a la detención remota de servicios, las solicitudes de detención se deben canalizar a través de los *proxies* de los servicios. Siguiendo esta premisa, los *proxies* de los servicios deben implementar el método `stopRemotely(ServiceID id)`, que solicita de manera remota a la clase *Server* la detención del servicio. Es necesario proporcionar al *Server* el `ServiceID` del servicio ya que éste no lo contiene, al tratarse de la clase en la que tiene lugar el procesado del servicio, y no su gestión, mantenimiento y publicación. La clase *Server* es capaz de lanzar eventos a la instancia de *CAEAT Service Manager* que la está manteniendo, por lo tanto redirecciona a éste la solicitud de detención.

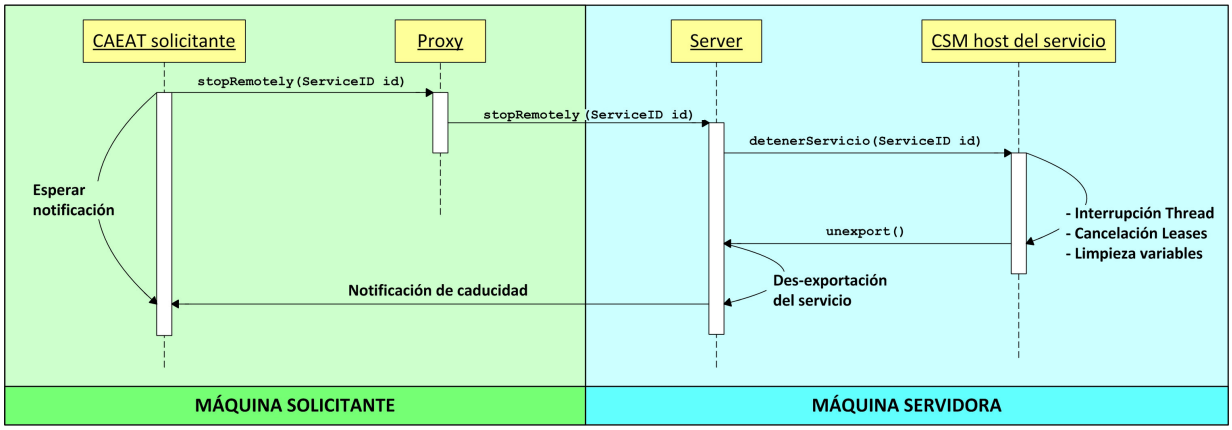

Figura 6.2: Secuencia de llamadas que intervienen en la detención remota de un servicio

Al recibir la solicitud, la única operación realizada por CSM es la interrupción al *Thread* que se encargaba de mantener activo el servicio y renovar periódicamente su *Lease* (ver capítulo 4 para más información). Este *Thread* mantenedor del servicio, en su gestión de la interrupción, es el encargado de realizar las tareas de limpieza: se limpian algunas variables internas de CSM relativas al servicio (índices, tiempos de vida, etc.), se solicita la cancelación de todos los *leases* asociados al servicio y en última instancia se des-exporta la clase *Server* para provocar que deje de atender llamadas remotas.

Se debe destacar que en caso de tratar con una agregación de servicios, la des-exportación y cancelación de los *leases* no se limita únicamente a la de la agregación, sino a la de todos los sub-servicios que "cuelgan" de ella. Como paso previo a su des-exportación, la clase *Server* notifica su caducidad a todos los clientes de su lista. Entre ellos se encontrará el cliente que ha realizado la solicitud (el propietario), que ha restado a la espera de dicha notificación como confirmación de la correcta detención del servicio. Si tras unos segundos de espera la notificación no llega, se informa al usuario de que posiblemente haya tenido lugar un error durante la detención. En caso contrario, el servicio recién detenido se elimina del tapiz.

Nótese que en el diagrama mostrado anteriormente se hace distinción explícita entre los ámbitos de la máquina que solicita la detención y la de la máquina que aloja el servicio. En el caso particular de que ambas máquinas sean la misma, el proceso de detención del servicio es el mismo para no perder generalidad.

Pese a que la forma más cómoda de proceder para conseguir la detención de un servicio consiste en el menú contextual de los componentes del tapiz, también es posible solicitar su detención desde la ventana de visualización de los servicios publicados en la máquina local. Esta interfaz fue presentada en el capítulo 4 y se puede observar más adelante en el presente capítulo (figura 6.11).

6.2. CAMBIO EN LA FECHA DE CADUCIDAD DE UN SERVICIO

Otra operación especial sobre el ciclo de vida de los servicios, menos drástica que la detención de los mismos, consiste en el cambio de la fecha inicial de caducidad que se fijó durante la publicación del mismo. La fecha de caducidad debe poder ser modificable por parte del creador del servicio ya que en un entorno como el propuesto por CAEAT / *SeNetComponents* se puede desear extender el ciclo de vida de un servicio (por ejemplo, porque está siendo muy utilizado por los clientes) o precipitar su expiración (por ejemplo, porque se está desarrollando su sustituto).

Los requisitos a cumplir por una operación como la descrita son básicamente dos:

- El propietario del servicio debe poder llevarla a cabo remotamente, desde cualquier punto de la red, sin necesidad de trabajar sobre la máquina que aloja el servicio.
- Al contrario que en el caso de la detención, los usuarios deben permanecer ignorantes al cambio de fecha de caducidad. La verdadera fecha de caducidad de un servicio es una información reservada al creador y publicador del mismo.

6.2.1. Implementación

La lógica de notificaciones y llamadas remotas que se debe llevar a cabo para conseguir el cambio de fecha de caducidad de un servicio es idéntica a la de su detención. A través del *proxy* se hace llegar al servidor la solicitud de cambio de fecha, acompañada de dos datos: el `ServiceID` del servicio y la nueva fecha de caducidad en milisegundos de acuerdo al formato aceptado por la plataforma Java.

La clase *Server* redirecciona la orden a su *CAEAT Service Manager*. Como se explicó en capítulos anteriores, CSM guarda una lista con los tiempos de vida de los servicios. Esta lista es consultada por los *Threads* que mantienen con vida los servicios para determinar si deben solicitar a los servidores LUS una renovación de los *leases* o simplemente dejarlos caducar. Esto significa que un cambio en la fecha de caducidad de los servicios puede implementarse de manera sencilla como un cambio en el tiempo de vida almacenado en dicha lista.

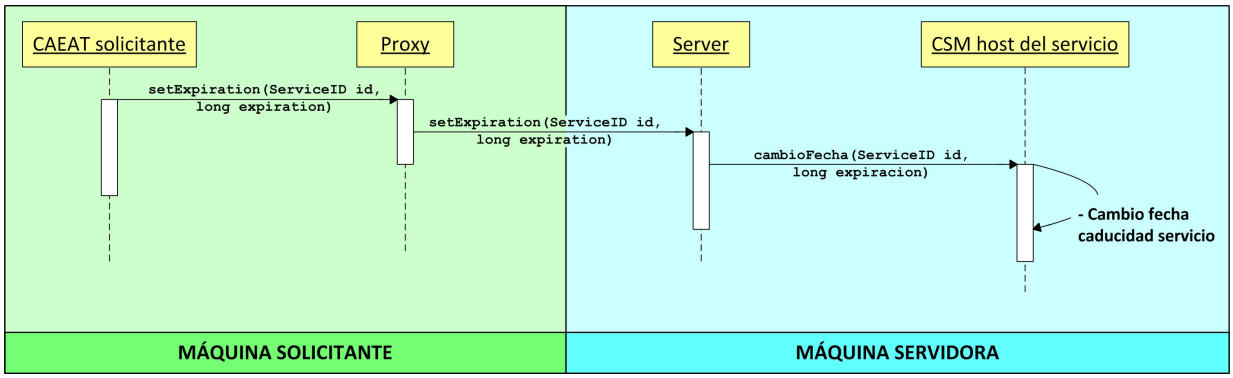

Figura 6.3: Secuencia de llamadas implicadas en el cambio de fecha de caducidad de un servicio

Para el cambio de fecha de caducidad, la máquina que aloja el servicio no devuelve ningún *feedback* de confirmación al solicitante del cambio. Se asume que los únicos errores que pueden surgir en una operación tan sencilla como un simple intercambio de valores son los asociados a la naturaleza distribuida de las llamadas remotas a través de la red. Estos errores podrán ser detectados desde el lado solicitante del cambio en caso de lanzarse una `RemoteException`.

6.2.2. Interfaz de usuario

Al contrario que en el caso de la detención del servicio, en la que no era necesaria más información por parte del usuario que la propia solicitud de detención, en este caso se requiere un *input* por su parte: la nueva fecha de caducidad del servicio. Para ello, se ha diseñado la interfaz de usuario que se muestra a continuación:

Figura 6.4: Interfaz de cambio de fecha de caducidad de un servicio

La interfaz de usuario hace uso del mismo calendario que se utilizaba para la selección de la fecha de caducidad en el momento de la publicación. La interfaz garantiza que la nueva fecha seleccionada sea una fecha posterior a la del día presente (que sea una fecha futura). Además, informa al usuario de la operación que está realizando sobre el servicio: un estiramiento o un acortamiento de su tiempo de vida, así como de la fecha de caducidad actual del servicio. Es posible marcar el servicio como permanente si no lo era ya, o fijar una fecha determinada en caso de que fuese permanente con anterioridad. Si la fecha introducida es correcta y distinta de la actual, la solicitud de cambio se realiza al pulsar “Aceptar” sobre esta ventana.

6.3. REINICIO DE UN SERVICIO

Una tercera operación especial sobre el ciclo de vida de los servicios representa un híbrido entre su detención y su cambio de fecha de caducidad. Es posible que en algún momento durante el ciclo de vida del servicio se desee retornar éste al estado inicial que presentaba en el momento inmediatamente posterior al de su publicación (sin clientes que hiciesen uso de él y con sus variables de proceso puestas a su estado inicial), y adicionalmente escogiendo una nueva fecha de caducidad.

El proceso descrito equivaldría a la detención completa del servicio y a su posterior re-inicialización y re-publicación. Sin embargo, ésta sería una forma redundante de proceder, puesto que se des-exportarían y des-publicarían una serie de objetos para a continuación volverlos a publicar y exportar. La operación de reinicio del servicio surge como una alternativa más eficiente a este procedimiento, pues aprovecha el hecho de que los objetos ya están publicados y exportados y que únicamente se deben llevar a cabo tres operaciones más para completar el reinicio:

- Expulsión del servicio de todos los clientes que estuviesen haciendo uso de él en el momento del reinicio, excepto del cliente que solicita el reinicio
- Restablecimiento de las variables de proceso del servicio a sus valores originales
- Establecimiento de una nueva fecha de caducidad (operación que hace uso de la lógica ya implementada en el apartado anterior)

6.3.1. Implementación

La lógica de llamadas necesaria para ejecutar el reinicio de un servicio es muy similar a la de los casos anteriores, por lo que únicamente se comentarán las diferencias:

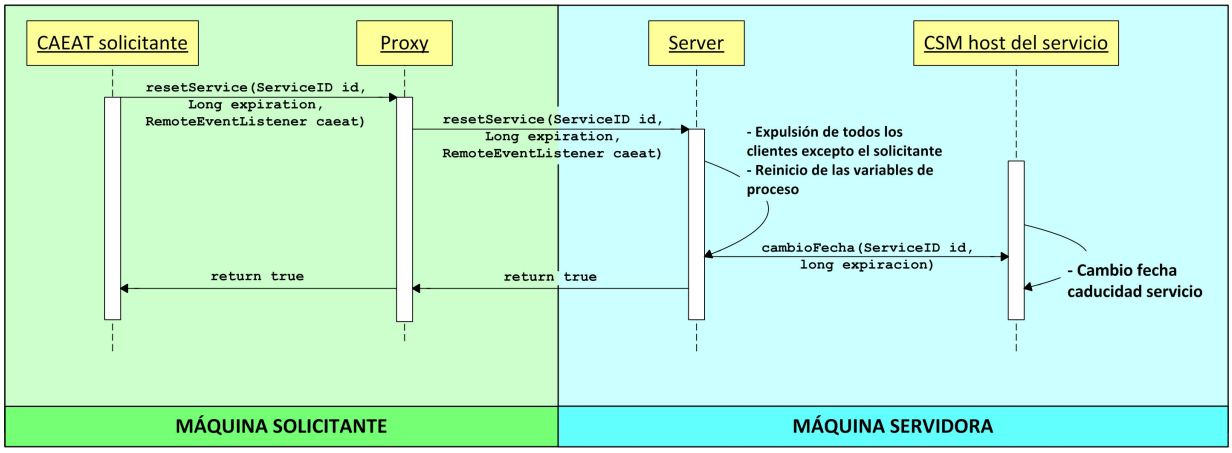

Figura 6.5: Secuencia de llamadas que intervienen en el reinicio de un servicio

La principal diferencia radica en el suministro a la clase *Server* de un `RemoteEventListener` que representa a la instancia de CAEAT que solicita el reseteo del servicio. Esta operación sirve para evitar que el *Server*, en su limpieza de la lista de clientes, elimine también al propietario del servicio, que pasaría a visualizar el mismo como caducado y debería volverlo a buscar en la paleta.

El *Server* expulsa a todos los clientes (menos al ya mencionado propietario) mediante el lanzamiento de un evento de caducidad del servicio. A continuación devuelve todas sus variables de proceso al estado inicial. El último paso consiste en la comunicación a CSM de la nueva fecha de caducidad, operación análoga a la del apartado anterior. En este caso, el *Server* sí devuelve al solicitante del reseteo una confirmación del éxito de la operación.

6.3.2. Interfaz de usuario

De nuevo, es necesario un *input* por parte del propietario del servicio que decide reiniciarlo: la nueva fecha de caducidad. Se ha diseñado para ello una interfaz gráfica ligeramente distinta que la del anterior apartado:

Figura 6.6: Interfaz de reinicio de los servicios

La interfaz permite escoger la nueva fecha de caducidad mediante el calendario, garantizando que ésta represente una fecha futura. De la misma manera que en el momento de la publicación, se permite el registro permanente del servicio. Se advierte al usuario que se procederá a la expulsión de los clientes del servicio. Si la fecha introducida es correcta, la solicitud de reinicio se lleva a cabo al pulsar sobre el botón verde de la presente ventana.

6.4. DES-PUBLICACIÓN DE UN SERVICIO

La última y más compleja de las operaciones especiales sobre el ciclo de vida de lo servicios consiste en la retirada de los servidores de *Lookup* de los objetos públicos que identifican a los mismos. Esta nueva funcionalidad responde a una necesidad por parte del usuario creador y publicador del servicio: es probable que, una vez lanzado el servicio, se desee por alguna razón hacer a éste invisible para los clientes. Sin embargo, se puede dar la situación de que no sea posible la detención del servicio (porque se trate de un servicio crítico que debe seguir ejecutándose) ni su reinicio (porque sea necesaria la conservación de los atributos de proceso del servicio).

Aunque ya se implementó un mecanismo de edición en caliente de agregaciones remotas, es posible que las modificaciones que se deban realizar sobre una agregación sean de tal magnitud que el usuario propietario desee realizar tales modificaciones “en privado”, sin la presencia de clientes remotos, para volver a hacer público el servicio una vez finalizado el trabajo. La operación anterior se debería poder llevar a cabo sin la necesidad de detener el servicio.

En situaciones como las descritas, la única alternativa es la implementación de un mecanismo de des-publicación del servicio que cumpla las siguientes características:

- La des-publicación debe suponer la retirada de los servidores de Lookup de los objetos del servicio, pero éste debe seguir activo y operando normalmente en la máquina en la que ha sido desplegado.
- Como excepción al resto de operaciones especiales de control de flujo, la des-publicación no podrá ser llevada a cabo de manera remota: el propietario del servicio deberá encontrarse trabajando en la misma máquina en la que el servicio se encuentra desplegado. Esto se debe a que la des-publicación implica la desaparición del *proxy* del servicio de la red. El propietario del servicio, en consecuencia, no será capaz de localizar su propio servicio en la federación si

lo des-publica desde una máquina remota, perdiendo todo control sobre él hasta acudir físicamente a la máquina que lo está ejecutando.

- Un servicio des-publicado debe seguir siendo activo en la máquina en la que está siendo ejecutado hasta el punto de poder ser insertable en el tapiz de CAEAT, y de poder interactuar con otros componentes del tapiz, ya sean *beans* locales o servicios remotos. El usuario debe ser capaz de distinguir fácilmente los servicios des-publicados de los servicios cuyo estado de publicación es el habitual.
- Un servicio des-publicado podrá ser re-publicado a petición del usuario en cualquier momento, pudiendo elegir nueva fecha de caducidad y nuevos grupos de registro. También debe ser re-publicado automáticamente si se publica todo el contenido del tapiz como un nuevo servicio y éste contiene un servicio des-publicado.
- Ante la des-publicación de una agregación, como en el caso de la detención, únicamente se des-publican los sub-servicios asociados a dicha agregación, no los servicios autónomos ajenos a ella que ésta pudiese contener.

6.4.1. Implementación

Como se ha comentado, la solicitud de des-publicación no se puede llevar a cabo de manera remota, por lo que ésta se realiza directamente sobre la instancia de *CAEAT Service Manager* ejecutándose en la máquina local.

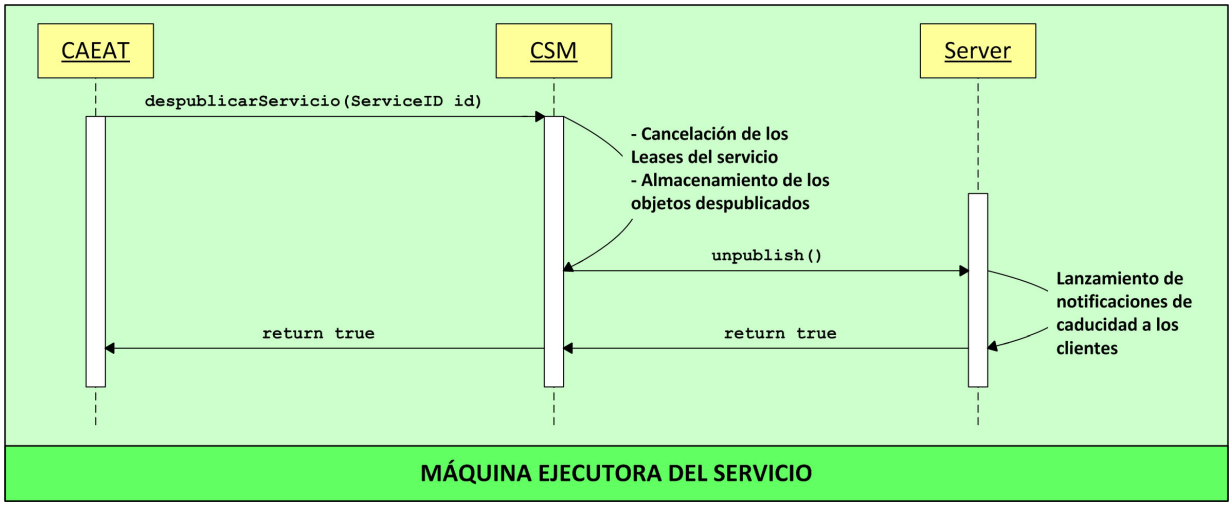

Figura 6.7: Secuencia de llamadas que intervienen en la des-publicación de un servicio

CAEAT ordena a su instancia de CSM la des-publicación del servicio cuyo identificador `ServiceID` le pasa como parámetro, y resta a la espera de una confirmación. CSM localiza el servicio afectado y cancela los *leases* de todos los objetos implicados en él (también el de los sub-servicios en caso de tratarse de una agregación). A continuación guarda una referencia de todos los objetos des-publicados. Esto es necesario por dos motivos: poder re-publicarlos rápidamente en el momento en que sea necesario, y permitir la utilización de los servicios en el tapiz local de CAEAT (los objetos se tomarán del CSM en lugar de obtenerlos de los servidores LUS repartidos por la red).

La última operación a realizar, que se lleva a cabo para todos los servicios y sub-servicios des-publicados, consiste en la notificación a la clase *Server* de dicha des-publicación. Nótese que la clase *Server* permanece inalterada, ya que es el lugar en el cual se lleva a cabo el procesado real del servicio y éste debe permanecer activo. Sin embargo, se debe informar de la desaparición del servicio a todos los clientes mediante una notificación de caducidad.

Nótese que no se ha empleado el término “expulsión”, ya que en este caso el *Server* no limpia y reinicia su lista de clientes. Ésta permanece inalterada hasta el momento de la re-publicación, consiguiendo de esta manera que todos los clientes que estuviesen haciendo uso del servicio en el momento de la des-publicación puedan seguir recibiendo eventos por parte de él una vez re-publicado (siempre que los clientes sigan estando activos y suscritos al servicio).

El proceso de re-publicación resulta mucho más sencillo, y no difiere en exceso del proceso de publicación de un servicio “desde cero”. CAEAT ordena a CSM que realice una búsqueda de los servidores LUS correspondientes a los grupos de registro que ha escogido el usuario. A continuación le ordena la re-publicación del servicio cuya clase publicadora se le proporciona como parámetro junto al nuevo tiempo de vida.

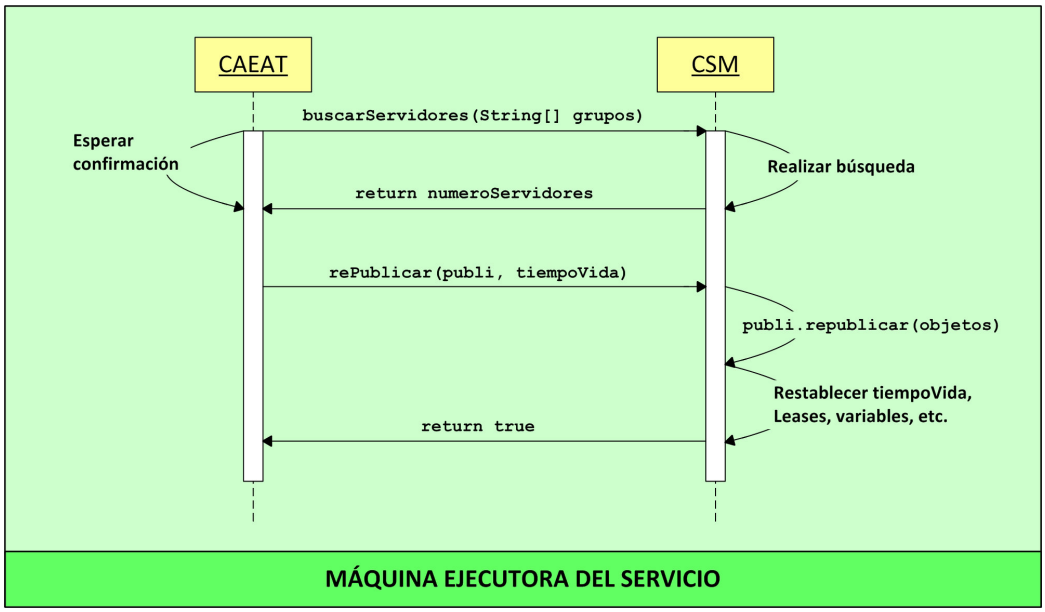

Figura 6.8: Secuencia de llamadas que intervienen en la re-publicación de un servicio

Dado que el objeto *Publi* encapsula el `ServiceID` del servicio, CSM es capaz de localizar los objetos públicos del servicio (que había almacenado durante el proceso de des-publicación) y re-publicarlos. Es extremadamente importante que en la re-publicación se proporcione a los servidores de Lookup el mismo `ServiceID` que el servicio estaba utilizando anteriormente a su des-publicación. Si se permite a los servidores LUS la asignación de un `ServiceID` nuevo, el servicio re-publicado será a todos los efectos un servicio completamente nuevo en la federación *Jini / Apache River*, y así lo entenderá también CAEAT: todos los servicios que referencien a éste en su documento XML dejarán de ser reconstruibles.

Por último, CSM restablece las variables de control: añade los nuevos *leases* a la lista de concesiones gestionadas, añade el nuevo tiempo de vida a la lista de servicios, etc. Si todas las operaciones resultan exitosas devuelve confirmación a CAEAT.

6.4.2. Cambios en la interfaz de CAEAT

La interfaz de usuario de CAEAT ha necesitado cambios puntuales para poder adaptarse a los requisitos de la nueva función de des-publicación. El primero de ellos ya fue introducido en el capítulo 5.3 cuando se habló del aspecto de los componentes sobre el tapiz: cuando un servicio se encuentra des-publicado se muestra en su lado inferior un icono específico. Además, la entrada del menú contextual en la que habitualmente se mostraba la opción de “Des-publicar” ahora mostrará la de “Re-publicar”:

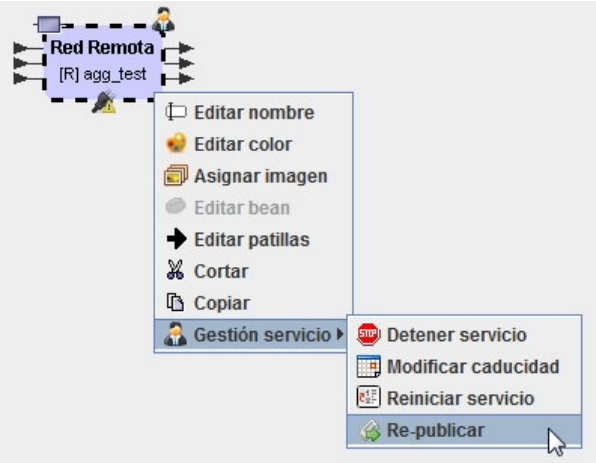

Figura 6.9: Menú contextual de operaciones de gestión de un servicio des-publicado

Los servicios des-publicados son utilizables en el tapiz, por lo que deben ser accesibles mediante la interfaz de usuario de CAEAT. Para tal fin se ha reservado un apartado especial en la paleta de servicios remotos. Cada vez que la paleta de servicios remotos es refrescada, se comprueba si en la máquina local existen servicios des-publicados. De ser este el caso, se agrega a la paleta remota un nuevo grupo de servicios llamado “NO PUBLICADOS”, claramente diferenciado de los grupos de *Lookup* encontrados mediante su iconografía.

La inserción de estos servicios en el tapiz es posible gracias a que *CAEAT Service Manager* guarda referencias (copias) de todos los objetos des-publicados y CAEAT las consulta en el momento de la inserción. La búsqueda de los objetos en los servidores LUS resultaría infructuosa ya que dichos objetos se han retirado de ellos en el momento de la des-publicación.

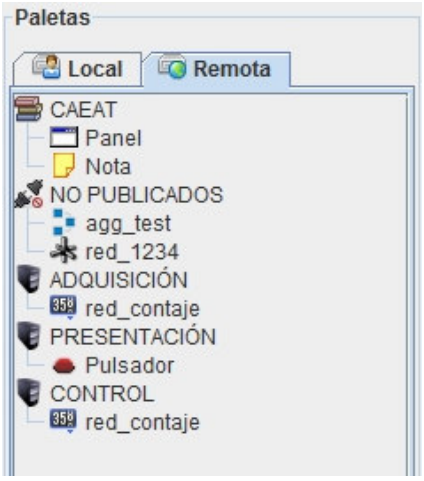

Figura 6.10: Paleta remota de CAEAT con servicios des-publicados en la máquina local

Por último, cabe destacar que la ventana de visualización de los servicios publicados en la máquina local ofrece una columna adicional de información sobre el estado de publicación de cada servicio (“publicado” o “activo des-publicado”), junto con un icono descriptivo de dicho estado. Desde esta misma ventana también es posible cambiar el estado de publicación de un servicio mediante su selección y la pulsación del botón “Publicar / Des-publicar”.

Figura 6.11: Interfaz de gestión de los servicios publicados desde la máquina local

6.5. SERVICIOS VIRALES

En una arquitectura orientada a servicios como el entorno *Jini / Apache River* las entidades participantes pueden unirse y abandonar la federación de servicios de manera dinámica y sin afectar el funcionamiento global de la federación. En el capítulo 1, en el cual se presentó la plataforma *Jini*, se mostró cómo este tipo de arquitecturas promueven la diversidad de roles (una misma entidad puede actuar de cliente, proveedor de servicios y servidor de *Lookup* al mismo tiempo si así lo desea), y la sencillez con la que entidades nuevas pueden iniciar sus servicios y publicarlos en la red, dándolos a conocer inmediatamente al resto de entidades.

Una de las consecuencias de esta característica, una de las más interesantes de las arquitecturas SOA, radica en la posibilidad de ampliar fácilmente y "en caliente" la infraestructura de la red mediante la inserción de nuevos servidores de *Lookup*. Es posible iniciar nuevos servidores LUS pertenecientes a grupos de registro ya existentes, para incrementar la redundancia que se le da a ese grupo. Es posible, del mismo modo, iniciar servidores LUS que constituyan un grupo de registro totalmente nuevo. Lógicamente, es igualmente posible que algunos servidores LUS desaparezcan por algún motivo (desaparición planificada, fallo de hardware, etc.), y la federación de servicios debe ser lo bastante robusta como para tratar estas eventualidades.

Bajo el esquema de publicación y mantenimiento de los servicios de CAEAT / *SeNetComponents* expuesto hasta el momento, esta característica plantea una disyuntiva importante, puesto que se debe decidir la forma de tratar los servidores de *Lookup* que puedan aparecer / desaparecer de la federación de servicios. Para afrontar la naturaleza cambiante de la red se ha diseñado e implementado el concepto de "servicio viral".

6.5.1. Problemática

El esquema de publicación diseñado es estático en lo que se refiere a los servidores de *Lookup*. Esto significa que los objetos públicos (*proxies*, *wrappers*) registrados en un determinado número de servidores LUS en el momento de la publicación permanecen publicados en esos mismos servidores durante todo el ciclo de vida del servicio, hasta su detención, sin expandirse a otros nuevos servidores de *Lookup* que pudiesen aparecer en la red a posteriori. En la práctica, esto puede llegar a inutilizar el servicio si se dan las circunstancias que se describen en la siguiente figura:

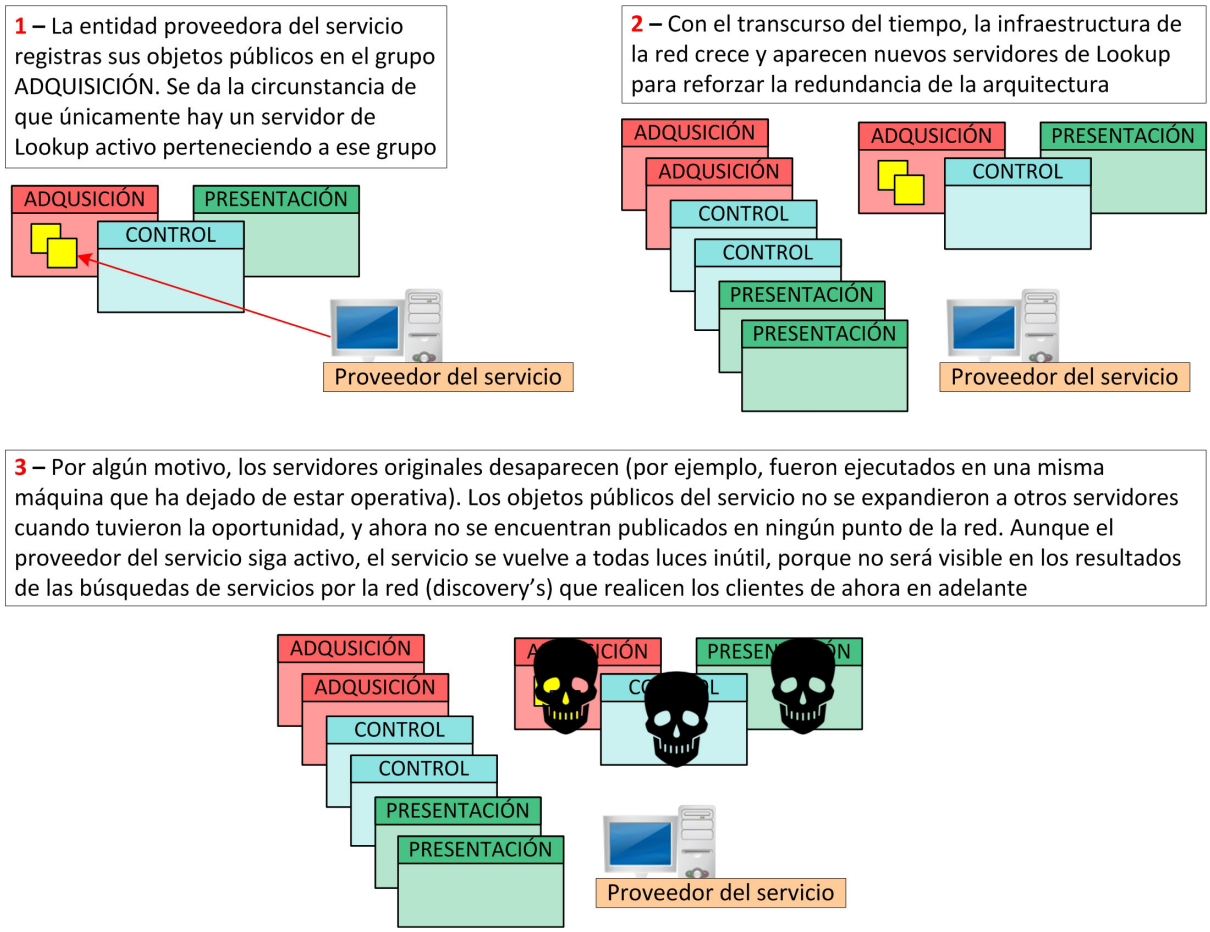

Figura 6.12: Problemática de supervivencia de los servicios no virales

Si el servicio fuese capaz de detectar la aparición de los nuevos servidores LUS pertenecientes al grupo "ADQUISICIÓN" y registrarse en ellos automáticamente, su supervivencia estaría asegurada. En el entorno CAEAT / *SeNetComponents*, los servicios capaces de auto-expandir sus objetos públicos de esta manera se han denominado "servicios virales".

6.5.2. Expansión de los servicios

Como proceso encargado del mantenimiento y supervivencia de los servicios, el trabajo de detección de la aparición y desaparición de servidores de *Lookup* en la red recae al 100% sobre *CAEAT Service Manager*. CSM dispone de un *Thread* que ejecuta dicho trabajo de manera periódica, con una periodicidad que el usuario puede escoger mediante el cuadro de selección opciones.

La lógica diseñada gira alrededor del correcto uso del objeto `ServiceRegistrar`. Este objeto es devuelto por los servidores de *Lookup* en el momento de realizar un *discovery*. Es el objeto que define e identifica al servidor LUS e implementa los métodos remotos que permiten registrar objetos en él. También permite diferenciar servidores distintos mediante una sencilla comparación de sus objetos `ServiceRegistrar`. En base a este hecho, el proceso de expansión de los servicios es el que se ilustra en la figura siguiente:

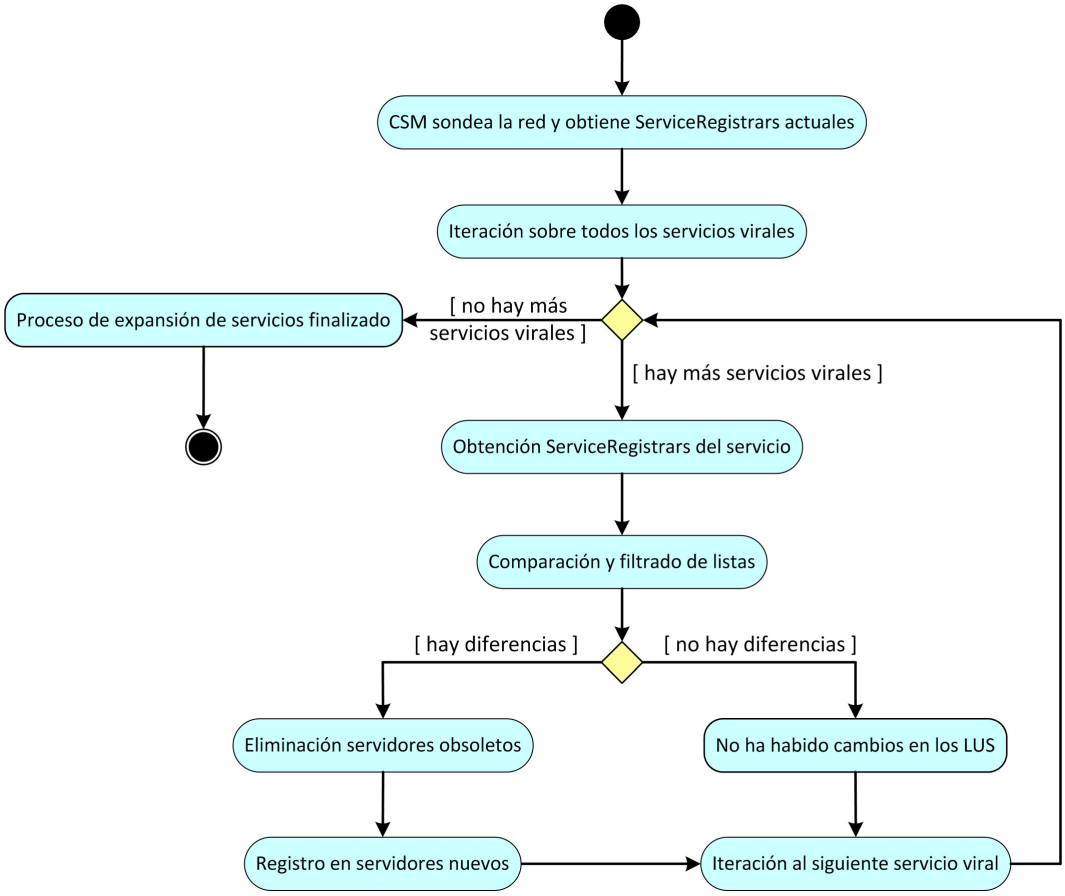

Figura 6.13: Proceso de expansión de los servicios virales

Cada servicio viral gestionado por la instancia de CSM guarda una lista fresca con los objetos `ServiceRegistrar` de los servidores de *Lookup* en los cuales se encuentra actualmente publicado. Cada vez que el *Thread* encargado de la expansión de los servicios se activa, realiza un sondeo rápido de la red para obtener los objetos `ServiceRegistrar` de todos los servidores de *Lookup* activos en ese momento. Con esa información, se itera sobre todos los servicios virales y se llevan a cabo las siguientes operaciones:

1. Se consulta al servicio para obtener la lista con los `ServiceRegistrar` que indica los servidores en los que se encuentra registrado el servicio en ese momento. Se consultan también los nombres de los grupos de *Lookup* en los que el servicio está registrado. Este último dato puede estar asignado a la constante `LookupDiscovery.ALL_GROUPS`, indicando que el servicio desea ser registrado en todos los grupos independientemente de su nombre.
2. Se compara la lista anterior con la lista de servidores LUS activos en el presente momento. Se elabora una lista de servidores desaparecidos si se detecta la falta de alguno de ellos en la lista recién obtenida mediante el *discovery*. Se elabora una segunda lista conteniendo los servidores LUS nuevos, cuyos objetos `ServiceRegistrar` no eran conocidos por el servicio. En esta segunda lista se filtran los servidores que no son de interés del servicio (que pertenecen a grupos de *Lookup* en los que el servicio no está interesado).
3. Se proporcionan ambas listas al servicio. Éste elimina de su lista los servidores LUS caducados y registra sus objetos públicos en los nuevos servidores encontrados, añadiendo éstos a una lista que mediante este mecanismo estará permanentemente fresca y actualizada. Nótese que el proceso de registro es inmediato y no requiere de ninguna otra búsqueda por parte del servicio, pues se le proporcionan directamente los objetos `ServiceRegistrar`, que ya encapsulan los detalles de comunicación con los servidores LUS.

6.5.3. Ejemplos y casuísticas

Se ofrecen en este apartado los diferentes casos que se pueden dar en el registro de los servicios en lo que respecta a su propiedad de viralidad. La característica de viralidad de un servicio se puede escoger en el momento de su publicación mediante un *checkbox* en la interfaz de usuario.

- **Servicio no viral**: es el tipo de registro que se ha asumido en el presente proyecto hasta este momento. El usuario escoge el conjunto de grupos de registro en los que desea que el servicio tenga presencia, y éste se registrará en todos los servidores LUS pertenecientes a esos grupos con presencia en ese momento en la red. Si en el futuro aparecen nuevos servidores LUS (ya sea pertenecientes a alguno de los grupos elegidos o pertenecientes a un grupo de registro nuevo), éstos no serán atacados.
- **Servicio viral sobre un conjunto de grupos**: si se opta por marcar el servicio como viral en el momento de la publicación, éste se expandirá (por defecto) únicamente a los grupos que el usuario ha escogido en el momento de la publicación. Se debe destacar que la elección de todos los grupos disponibles en el momento concreto del registro significa que el servicio será viral únicamente sobre esos grupos, no sobre los nuevos que puedan aparecer en un futuro. Para tal fin, se ha habilitado el último de los casos.
- **Servicio viral sobre todos los grupos**: este último caso es particular y se debe activar explícitamente mediante un *checkbox* en la interfaz de usuario de publicación. Significa que el servicio será viral sobre todos los grupos, tanto los existentes como los que pudiesen aparecer en el futuro. En este caso es irrelevante la selección de un conjunto de grupos de registro, por lo que se desactivan los recuadros que sirven para tal fin.

Figura 6.14: Ejemplo de registro de servicio permanente y viral hacia todos los grupos existentes

En la práctica, los objetos públicos de un servicio viral sobre todos los grupos “infectarán” a todos los servidores de *Lookup* de la red independientemente del grupo de registro al que estén asociados, alcanzando máxima visibilidad y redundancia.

7. CAEAT NETWORK ACCEPTOR

El diseño e implementación del módulo de software que se ha denominado *CAEAT Network Acceptor* surge con la finalidad de ampliar enormemente las posibilidades y la versatilidad de la plataforma CAEAT / *SeNetComponents*. El esquema y la arquitectura de publicación y exportación de servicios presentada hasta ahora propone un modelo de despliegue y gestión de los servicios totalmente deslocalizado:

- Los servicios pueden ser obtenidos y utilizados desde cualquier punto de la red, independientemente del lugar real en el cual se estén ejecutando.
- Los propietarios de los servicios igualmente pueden (con alguna excepción) gestionar los servicios desplegados desde cualquier punto de la red. Es posible llevar a cabo la mayoría de operaciones de control del flujo de vida de los servicios (explicadas en el capítulo 6) de manera remota, sin importar la localización del usuario propietario respecto a la del servicio.
- Se promueve la diversificación de roles de las máquinas pertenecientes a la federación de servicios: una máquina que ofrece servicios puede perfectamente actuar de cliente de otros servicios, e incluso ejecutar una o más instancias de servidores de *Lookup*.

Sin embargo, este esquema a priori tan deslocalizado y descentralizado presenta una limitación crucial: el despliegue y exportación de servicios está limitado a la máquina local desde la cual el propietario del servicio decide iniciarlo y publicarlo. *CAEAT Network Acceptor* es un módulo de software, independiente a la plataforma de edición CAEAT pero estrechamente relacionado a ella, que nace con el objetivo de permitir el despliegue remoto de servicios. Esto significa que el creador, propietario y publicador de un servicio será capaz de inicializar éste en cualquier máquina de la red que esté preparada para ello, y no únicamente en la máquina en la que se encuentra operando físicamente.

7.1. OBJETIVO Y DEFINICIONES

En anteriores capítulos se ha presentado la máquina proveedora del servicio como aquélla en la que tiene lugar el procesado real del servicio. La máquina proveedora exporta una serie de clases del lenguaje de programación *Java* (que en las librerías implementadas se han denominado *Server*) que contienen y procesan las variables reales del servicio, y con las cuales los clientes pueden interactuar mediante llamadas remotas a través de un *proxy* que la máquina proveedora habrá publicado en los servidores de *Lookup* para darse a conocer a la federación.

Los conceptos de “publicar” y “exportar” un servicio están intrínsecamente ligados a la máquina o entidad de la red desde la cual el propietario del servicio está trabajando. Al realizar el proceso de publicación, el servicio se exporta en la máquina local, los recursos públicos del servicio se copian en la carpeta principal del servidor HTTP y los objetos públicos del servicio se registran en los servidores de *Lookup* desde la máquina local. La máquina desde la cual se ha publicado el servicio pasa a ser irremediablemente la máquina proveedora del servicio real.

El objetivo de *CAEAT Network Acceptor* es disponer a lo largo de la red de máquinas preparadas para actuar de proveedoras de servicios. El usuario creador de un servicio podrá optar por inicializarlo y publicarlo desde la máquina local (proceso descrito hasta el momento) o acceder remotamente a alguna de estas nuevas máquinas y desplegar y publicar el servicio desde ellas tal y como si se tratase de la máquina local. Un ejemplo de la finalidad práctica que se persigue con esta ampliación del software se muestra en la figura siguiente:

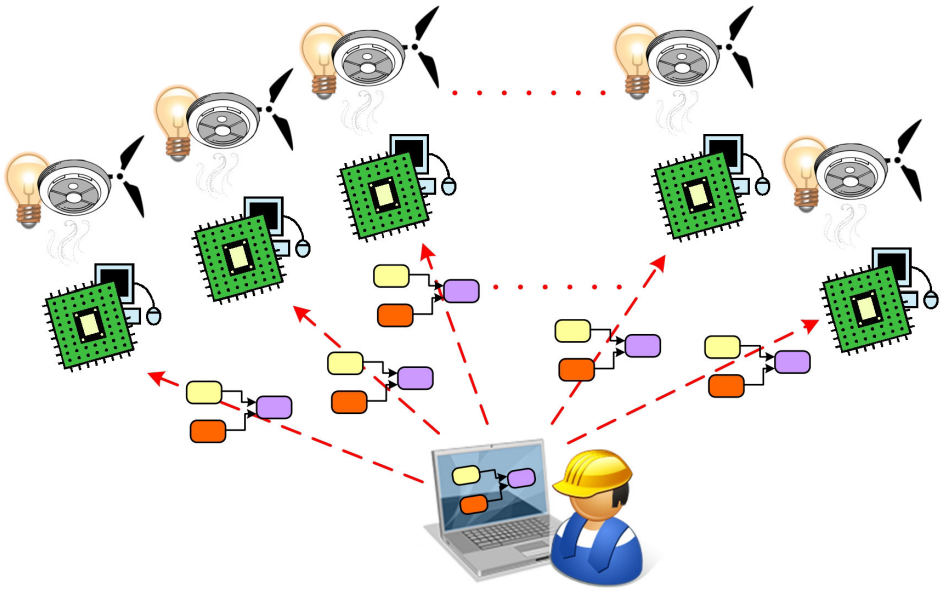

Figura 7.1: Escenario de aplicación de *CAEAT Network Acceptor*

Supóngase un escenario consistente en un número elevado de máquinas o entidades que controlan multitud de actuadores y elementos de campo. Es posible que se desee inicializar el mismo servicio en todas ellas (por ejemplo, porque la agregación de servicios que interactuaba con los elementos de campo hasta el momento ha quedado obsoleta). Un operario posee la agregación de componentes correspondiente al nuevo servicio (ya sea desde el entorno CAEAT o encapsulada en un archivo *jar* autoejecutable), y desea desplegar el nuevo servicio de inmediato sobre todas las máquinas.

Con el esquema de publicación diseñado hasta el momento, el operario encargado de la inicialización debe desplazarse físicamente a cada máquina objetivo, desplegar y publicar el servicio desde ella. Con *CAEAT Network Acceptor*, las máquinas controladoras de los elementos de campo estarán preparadas para recibir por parte del usuario todos los recursos relacionados con el servicio, pudiendo a continuación desplegarlo y publicarlo tal y como si el usuario estuviese operándolas de manera local. Contando con tal mecanismo, el operario será capaz de:

- Conectarse a la red formada por las máquinas en las que acabará residiendo el servicio y entrar en la federación de servicios a través de la ejecución de CAEAT o de un esquema autoejecutable encapsulado en un archivo *jar*.
- Sondear la red para obtener un listado de las máquinas que admiten el despliegue y publicación remota de agregaciones de servicios.
- Seleccionar algunas de las máquinas obtenidas (o todas) para el despliegue remoto de la agregación de servicios en ellas.
- Abandonar completamente la red al finalizar la operación, dejando a los nuevos servicios operando en sus correspondientes máquinas objetivo del mismo modo que si se hubiesen inicializado físicamente desde cada una de ellas.

Se consigue de esta manera desacoplar la ubicación física del usuario publicador de la agregación / servicio de la ubicación física de la máquina o entidad que actuará finalmente como proveedora del servicio.

7.2. REQUISITOS Y DISEÑO

CAEAT Network Acceptor (CNA) es un módulo de software totalmente independiente a CAEAT que se ejecuta como un servicio (es decir, permanentemente) en aquellas máquinas a las que se quiere dotar de la posibilidad de recibir agregaciones de servicios.

Este apartado recoge algunas decisiones de nomenclatura que se han adoptado para designar a las nuevas operaciones que es posible realizar con la plataforma CAEAT y CNA, enumera los requisitos de un módulo de software como CNA y detalla las estrategias de diseño adoptadas (basadas en la arquitectura *Jini / Apache River*) para conseguir los objetivos propuestos.

7.2.1. Definiciones

En adelante, se hará referencia a los siguientes términos y conceptos introducidos en el marco del desarrollo de CNA:

- **Máquina red-aceptante**: se define con este nombre a una máquina que está ejecutando una instancia de *CAEAT Network Acceptor*, y que por lo tanto está lista para recibir por parte de otras máquinas los recursos relacionados con las agregaciones de servicios e inicializar y publicar dichos servicios de la misma manera y con los mismos resultados que si se desplegaran operando manualmente sobre la propia máquina red-aceptante. El rol de máquina red-aceptante no es mutuamente excluyente con los demás roles desarrollados durante el presente proyecto: una máquina puede ser al mismo tiempo red-aceptante y estación de trabajo para CAEAT, o incluso alojadora de servidores de *Lookup*.
- **Lanzamiento de agregaciones**: se ha definido como “lanzamiento” a la acción de desplegar una agregación de servicios en una máquina red-aceptante remota (diferente de la máquina desde la cual se está “lanzando” el servicio). En adelante, el término “publicación” hará referencia a la publicación tradicional (la máquina desde la cual se publica será la proveedora del servicio), mientras que el término “lanzamiento” hará referencia a la definición recién expuesta.

7.2.2. Requisitos

Se presenta a continuación una lista con los requisitos que debe cumplir el diseño de *CAEAT Network Acceptor*.

1. CNA debe ser un programa completamente independiente a CAEAT que se ejecute permanentemente en las máquinas que se deseen marcar como red-aceptantes.
2. CNA y CAEAT deben ser totalmente compatibles entre sí en una misma máquina.
3. Las máquinas red-aceptantes darán a conocer su presencia al resto de integrantes de la federación de servicios mediante la publicación de un objeto *Proxy* en los servidores de *Lookup*.
4. Una vez exportado, publicado e iniciado el servicio, no debe haber diferencia funcional en su mantenimiento y gestión respecto de la de los servicios publicados mediante los canales disponibles hasta ahora (CAEAT, archivos autoejecutables).
5. Desde la máquina red-aceptante que ejecuta CNA debe ser posible listar los servicios actualmente publicados y detener manualmente los que se desee.
6. La finalización de CNA provocará que la máquina deje de ser red-aceptante. Previamente a su finalización se procederá a la detención ordenada de todos los servicios gestionados por esa máquina.

7. Debe ser posible inicializar máquinas red-aceptantes protegidas por contraseña para que únicamente un subgrupo de usuarios tenga acceso al lanzamiento de agregaciones de servicios hacia esa máquina.

7.2.3. Diseño

Dado que CNA debe gestionar y mantener los servicios lanzados desde el exterior de la misma manera que lo hace CAEAT, se ha decidido que este nuevo módulo de software haga uso de *CAEAT Service Manager* de la misma manera que lo hacía la plataforma de edición. De esta manera, *CAEAT Network Acceptor* estará compuesto por dos programas *Java* claramente diferenciados:

- **CSM**: el programa gestor de servicios que se lanza automáticamente cuando se desea publicar un servicio, y que se mantiene activo durante todo el ciclo de vida de los mismos garantizando su supervivencia y llevando a cabo las renovaciones de las concesiones de los servidores LUS correspondientes. Dado que se desea total compatibilidad entre los servicios publicados directamente desde CAEAT y los servicios lanzados a través de CNA, la versión de CSM usada en este nuevo módulo se software es idéntica a la utilizada por la plataforma de edición, cuyo funcionamiento se detalló en profundidad en el capítulo 4.
- **CNA**: *CAEAT Network Acceptor*, por lo tanto, es un programa autónomo que actúa como "puente" entre las instancias de CAEAT repartidas por la red y la de CSM que opera en la máquina red-aceptante local. CNA se exporta como servicio y se publica su *proxy* en los servidores LUS correspondientes, de manera que otras instancias de CAEAT puedan interactuar con él a través de llamadas remotas cada vez que deseen lanzar un nuevo servicio a la máquina red-aceptante. La siguiente figura ilustra esta arquitectura:

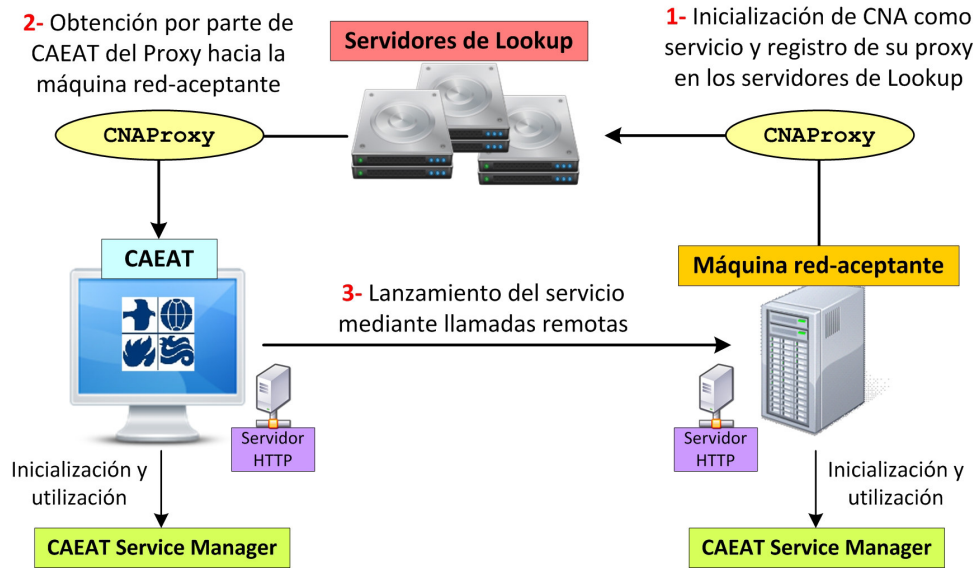

Figura 7.2: Arquitectura de proceso implementada para el *CAEAT Network Acceptor*

Se puede observar cómo CAEAT puede decidir publicar su servicio en la máquina local, para lo cual lanzaría una instancia de CSM en su entorno local y procedería de la manera que se detalló en capítulos anteriores; o puede decidir lanzar el servicio a una máquina red-aceptante, para lo cual se obtendría primero su *proxy* desde los servidores de *Lookup* para a continuación iniciar el proceso de lanzamiento mediante llamadas remotas (dicho proceso se detalla en el siguiente apartado).

Las funciones de CNA, sin embargo, no se limitan a las de un mero "puente" entre clientes remotos y una instancia del gestor CSM. También proporciona la lógica y las interfaces de usuario necesarias para obtener sus datos de inicialización (nombre, descripción, contraseña, etc.) y exportar y publicar el propio CNA como servicio. Nótese que la instancia de CSM de una máquina red-aceptante que aún

no ha recibido ningún servicio desde el exterior gestionará como mínimo un servicio: el del propio CNA, que por defecto se publica sin fecha de caducidad y se debe mantener con vida hasta que sea finalizado de manera explícita (es decir, hasta que un usuario decida que la máquina red-aceptante debe dejar de serlo).

CNA también ofrece interfaces gráficas para listar todos los servicios desplegados en la máquina desde la cual se está ejecutando, ya sean éstos servicios recibidos desde el exterior o desplegados desde la propia máquina mediante CAEAT. También es posible ocultar las interfaces gráficas de CNA dejando su ejecución en segundo plano: la máquina será red-aceptante hasta que no se detenga explícitamente el programa CNA.

Se ha impuesto como requisito la compatibilidad de CAEAT y CNA en una misma máquina. Esta compatibilidad se alcanza gracias al uso compartido que los dos módulos de software realizan del gestor de servicios CSM. La siguiente figura ilustra dicho uso compartido:

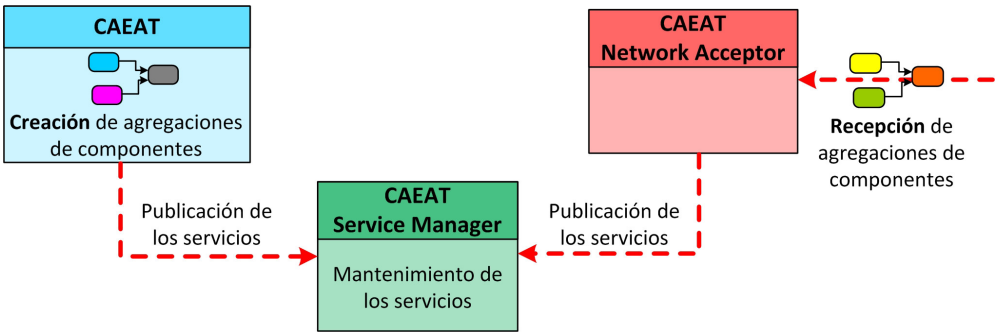

Figura 7.3: Uso compartido de CSM por parte de CAEAT y CNA

Cabe recordar que CSM hacía uso de las herramientas proporcionadas por el paquete RMI de *Java* para registrar una interfaz remota en un puerto conocido de la máquina local. CAEAT obtenía a posteriori dicha interfaz remota para poder gobernar CSM mediante llamadas remotas y para poder permitir la comunicación, colaboración e intercambio de objetos entre diferentes Máquinas Virtuales de *Java*.

La interfaz remota de CSM puede ser igualmente obtenida por CNA. La ilustración anterior ejemplifica una máquina en la cual CAEAT publica sus servicios desde la máquina local a través de CSM, tal y como se ha explicado hasta el presenta capítulo. Pero al mismo tiempo (y en segundo plano), es perfectamente posible y compatible que una instancia de CNA se esté ejecutando, y por lo tanto la máquina sea red-aceptante. CNA realiza un uso de CSM idéntico al que realiza CAEAT, con la diferencia de que los servicios que se desea publicar no son creados localmente, sino recibidos de manera remota procedentes de otras instancias de CAEAT ejecutándose por la red.

Nótese que en una situación como la mostrada en la figura anterior se tienen tres Máquinas Virtuales de *Java*, cada una ejecutando uno de los módulos de software distintos desarrollados en el marco del presente proyecto, que colaboran entre ellas con total compatibilidad (cabe recordar que las variables críticas de proceso del CSM son sincronizadas).

7.2.3.1. Interfaz pública de CNA

Como todo servicio *Jini*, CNA cuenta con una interfaz pública bien conocida por los clientes que define claramente los métodos remotos que es posible invocar sobre CNA como un servicio más. Esta interfaz es implementada por el objeto *Proxy* de CNA así como por el objeto *Server* que es exportado y que se encarga de atender las llamadas remotas a los distintos métodos. La interfaz pública de CNA se muestra a continuación:

```
public interface NetworkAcceptorInterface extends Remote {

        //Métodos que sirven de "puente" entre la máquina que está lanzando la red y la
        máquina aceptante:
        int getNumServicios() throws RemoteException;
        boolean detenerServicio(ServiceID id) throws RemoteException;
        boolean finalizaManager() throws RemoteException;
        ArrayList<PublicadorInterface> getListaPublicados() throws RemoteException;
        Object[] publicaBean(PublicadorInterface publi, Serializable server,
                        WrapperInterface wrapper, boolean unico) throws RemoteException;
        Object[] publicaXML(PublicadorInterface XMLPubli,
                        AggregationServer aggServ) throws RemoteException;
        boolean isMaquinaLocal(ServiceID id) throws RemoteException;
        int buscarServidores(String[] groups) throws RemoteException;
        void finalizaPublicacion(boolean errores, long tiempoVida) throws RemoteException;
        ArrayList<ServiceID> getIDSubservicios(ServiceID servicio) throws RemoteException;
        String getCodebase() throws RemoteException;

        //Métodos que ayudan a distribuir los datos de la máquina red-aceptante:
        String getNombre() throws RemoteException;
        String getDescripcion() throws RemoteException;
        URL getIcono() throws RemoteException;
        boolean isProtegida() throws RemoteException;
        boolean checkPassword(String passwordCliente) throws RemoteException;

        //Métodos propios de despliegue y gestión del CAEAT Network Acceptor:
        void setServerPath(String serverPath) throws RemoteException;
        void setManager(CAEATServiceManagerInterface csm) throws RemoteException;
        void setCNAListener(RemoteEventListener l) throws RemoteException;
        boolean copiarRecursosRemotos(ArrayList<URL> resources, String codebase) throws
        RemoteException;
        void actualizarInterfaz() throws RemoteException;
        void setServiceID(ServiceID id) throws RemoteException;
        ServiceID getServiceID() throws RemoteException;
        boolean isAlive() throws RemoteException;

}
```

Código 7.1: Interfaz pública de *CAEAT Network Acceptor*

Se puede observar que los métodos a implementar están claramente diferenciados en tres categorías:

- Métodos de idéntico nombre y estructura que los ofrecidos por *CAEAT Service Manager* para la publicación de servicios. Este grupo de métodos representa el ya citado "puente" entre los clientes externos y la instancia de CSM local. Por lo general, estos métodos simplemente delegan el trabajo en dicha instancia local del CSM.
- Métodos de distribución de datos de la máquina red-aceptante. Estos métodos se usan en todos los servicios especiales implementados (tanto CNA como los que se presentarán en el capítulo siguiente), y sirven para proporcionar a los usuarios información en formato amigable (nombre, icono, descripción) como resultado a las búsquedas realizadas.
- Métodos de despliegue y gestión del servicio (usados únicamente en el momento de su inicialización), así como métodos propios de CNA. Muchos de ellos, como `isAlive()`, son comunes a todos los servicios implementados durante la elaboración de este proyecto, sin embargo también se ofrecen métodos propios de la lógica de funcionamiento de CNA. Entre estos últimos se encuentra *copiarRecursosRemotos*, el método más importante ofrecido por CNA y que será explicado en más detalle en el apartado siguiente.

7.3. LANZAMIENTO DE SERVICIOS

A la vista de la interfaz remota diseñada y presentada en el apartado anterior, se puede deducir que el proceso de lanzamiento de un servicio no diferirá en exceso del de su publicación desde la máquina local. Nótese que la interfaz pública define multitud de métodos idénticos a los que se definían para gobernar *CAEAT Service Manager* (véase el capítulo 4). Estos métodos son llamados por CAEAT tal y como si la publicación estuviese teniendo lugar en la máquina local, pero en realidad se delega en llamadas remotas que acaban desembocando en el CSM de la máquina red-aceptante.

Existe, sin embargo, una particularidad en el lanzamiento de servicios: la máquina red-aceptante no tiene conocimiento alguno de las clases ni los recursos (iconos, imágenes, etc.) que conforman dicho servicio. Cuando se publica desde la máquina local, las clases a utilizar y recursos a compartir se encuentran encapsulados en el archivo *jar* de la librería correspondiente. Dado que la máquina que exporta el servicio (la máquina red-aceptante en este caso) debe compartir los recursos públicos del mismo permanentemente a través de su servidor HTTP, es necesario un mecanismo para ponerlos a su disposición.

La estrategia adoptada requiere que la máquina "lanzadora" (en la cual se ejecuta CAEAT) también disponga de un servidor HTTP, tal y como si se dispusiese a publicar el servicio por sí misma. Todos los recursos públicos se copian al directorio del servidor HTTP como si se tratase de una publicación normal, a la vez que se genera una lista de URL's a través de las cuales los recursos son accesibles "desde fuera". Como paso previo al lanzamiento del servicio, se proporciona a la máquina red-aceptante esta lista de URL's, pudiendo ésta copiar a su espacio local los recursos que le están "lanzando".

El diagrama de secuencia completo del proceso de lanzamiento de un servicio se puede observar a continuación:

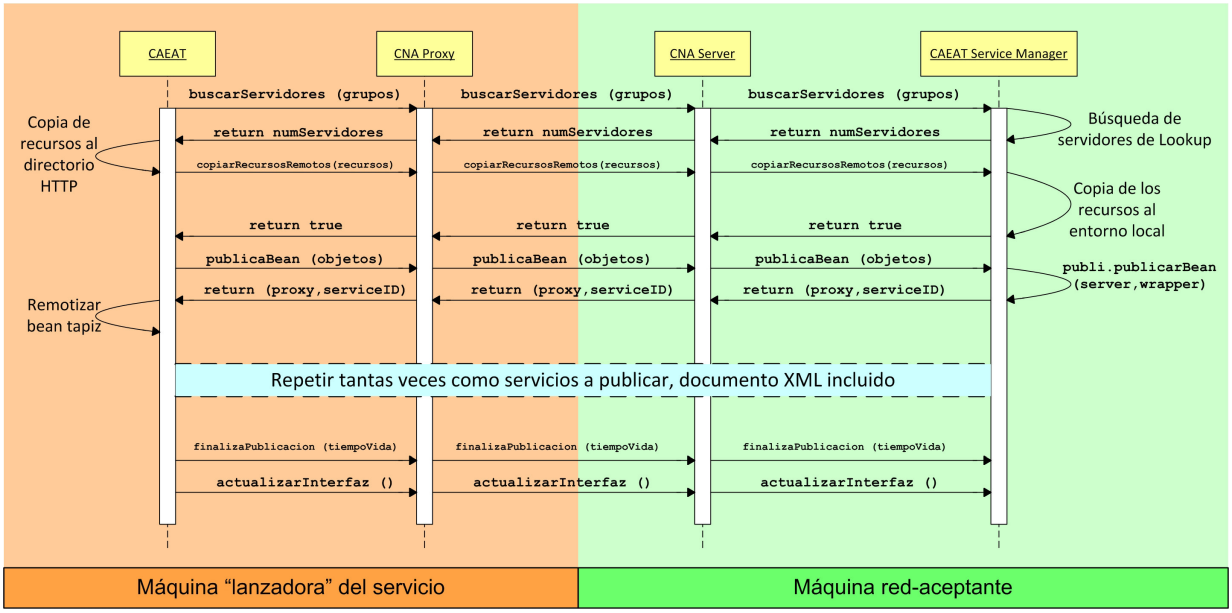

Figura 7.4: Diagrama de secuencia del lanzamiento de un servicio a una máquina red-aceptante

En primer lugar, CAEAT solicita mediante llamada remota a la instancia de CSM de la máquina red-aceptante la búsqueda y obtención de referencias a los servidores LUS de los grupos de registro objetivo. Tras recibir confirmación, CAEAT copia a la carpeta pública de su servidor HTTP todos los recursos necesarios para desplegar ese servicio (clases *Proxy*, *Server*, *Wrapper*, *BeanInfo*, *Customizer*, iconos, imágenes, otras clases necesarias, etc.). A su vez, genera una lista con las URL's que permiten obtener esos recursos desde el exterior. La lista es enviada a la máquina red-aceptante.

La máquina red-aceptante copia todos los recursos contenidos en la lista a la carpeta pública de su propio servidor HTTP, imitando la arborescencia de directorios creada por CAEAT y plasmada en las URL's. Este es el proceso que ralentiza en mayor medida el lanzamiento de un servicio complejo, por lo que CAEAT espera confirmación de la correcta copia mediante la recepción del *boolean true*.

El proceso que se lleva a cabo a continuación es exactamente el mismo que se realizaba en el caso de la publicación desde la máquina local, con la diferencia de que las llamadas a los métodos de publicación acaban desembocando en llamadas remotas hacia la máquina red-aceptante. En el caso de la publicación tradicional, las peticiones se realizaban directamente sobre la instancia de *CAEAT Service Manager* de la máquina local: en este caso se realizan sobre un *proxy* que las redirecciona hacia la máquina red-aceptante. Se publican de esta manera todos los *beans* o sub-servicios pertenecientes a la agregación de servicios así como el XML descriptor de la agregación.

Al finalizar la publicación, CAEAT envía a la máquina red-aceptante el tiempo de vida que el usuario ha escogido para el servicio. También se realiza una petición de actualización de la interfaz gráfica de la máquina red-aceptante, en el caso de que dicha interfaz esté siendo mostrada en el presente instante (ver apartado siguiente).

7.4. EJEMPLOS DE UTILIZACIÓN E INTERFAZ GRÁFICA

Se presenta en este último apartado un ejemplo de utilización, tanto desde la óptica de la máquina red-aceptante como desde la del cliente CAEAT que interactúa con ella y lanza servicios hacia ella. Se presentan asimismo las interfaces de usuario desarrolladas para poder manejar correctamente este nuevo módulo de software.

7.4.1. Inicialización de CAEAT Network Acceptor

Para marcar una máquina como red-aceptante es necesario ejecutar CNA y exportarlo como servicio, a la vez que se publica su *proxy* en los servidores LUS escogidos. En el momento de la ejecución de CNA, se comprueba si la máquina local ya es red-aceptante, es decir, si ya existe una instancia de CNA exportada y operando. En caso afirmativo el programa finaliza inmediatamente, puesto que no tiene ningún sentido iniciar una segunda instancia de CNA en una máquina que ya es red-aceptante. En caso negativo se muestra la interfaz de inicialización de CNA, que se muestra en la figura siguiente:

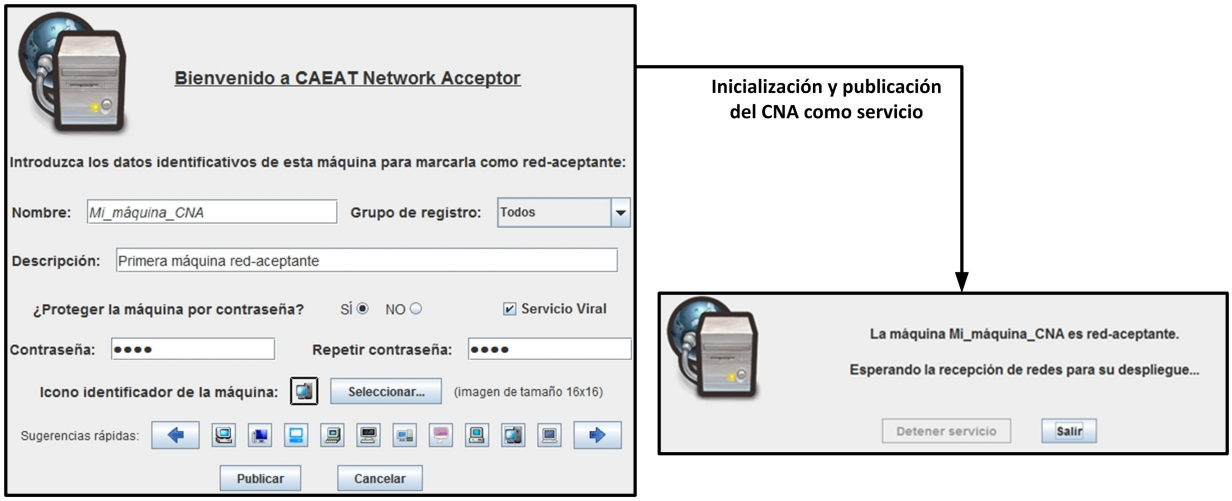

Figura 7.5: Interfaz de introducción de datos e inicialización de CNA

Al mismo tiempo se habrá comprobado también la presencia de servidores de *Lookup* en la red (como mínimo uno de ellos) y la presencia de un servidor HTTP en la máquina local (requisito imprescindible para poder publicar servicios). Si alguno de estos dos elementos no está presente, se muestra la interfaz de inicialización de todas formas, pero se advierte al usuario de que no se podrá marcar la máquina local como red-aceptante hasta que se arranquen ambos servidores.

La interfaz de inicialización permite establecer los datos descriptivos del servicio, como su nombre, su descripción y su icono asociado (que será compartido a través del directorio público del servidor HTTP). Estos datos serán mostrados por los clientes como resultado de las búsquedas. También es posible elegir un grupo de registro en el cual dar a conocer la máquina red-aceptante (usualmente se deseará dar máxima visibilidad a los servicios especiales como CNA), así como seleccionar si el *proxy* registrado en los servidores LUS debe ser viral y extenderse ante la aparición de nuevos servidores.

Por último, es posible proteger la máquina red-aceptante mediante el establecimiento de una contraseña. En caso de encontrarse protegida, la máquina será visible por los clientes de CAEAT en sus búsquedas, pero no será posible lanzar servicios hacia ella a menos que el usuario proporcione la contraseña correcta. Si la exportación y publicación del servicio es exitosa, se muestra una interfaz que informa al usuario de que la máquina pasa a ser red-aceptante desde ese momento. Esta interfaz será actualizada de manera dinámica cada vez que un cliente lance un servicio para mostrar una lista de los servicios actualmente gestionados por CNA.

7.4.2. Lanzamiento de un servicio desde CAEAT

Con tal de facilitar el acceso a los servicios especiales desarrollados alrededor de la plataforma de edición CAEAT, se ha creado la paleta “Servidores”; situada bajo la paleta de servicios (tanto locales como remotos) presentada en apartados anteriores. Esta paleta muestra las máquinas red-aceptantes encontradas a lo largo de la red, así como los servidores de imágenes y repositorios de librerías (objetos de estudio del capítulo siguiente). La paleta “Servidores” es poblada de servidores en el momento del arranque de CAEAT y periódicamente cada vez que se refresca la paleta de servicios remotos.

Realizar doble *clic* sobre la entrada de la paleta que representa una máquina red-aceptante inicia el proceso de lanzamiento del esquema que se tenga en ese momento en el tapiz hacia dicha máquina. Previamente se comprueba si la máquina está protegida por contraseña, en cuyo caso se pide al usuario que introduzca esa información. La contraseña real jamás llega a viajar a través de la red: la cadena introducida por el usuario es enviada a la máquina red-aceptante y es ésta la encargada de realizar la comprobación y devolver un valor de tipo *boolean* (*true* / *false*) para indicar si el usuario tiene permiso para lanzar la agregación de componentes.

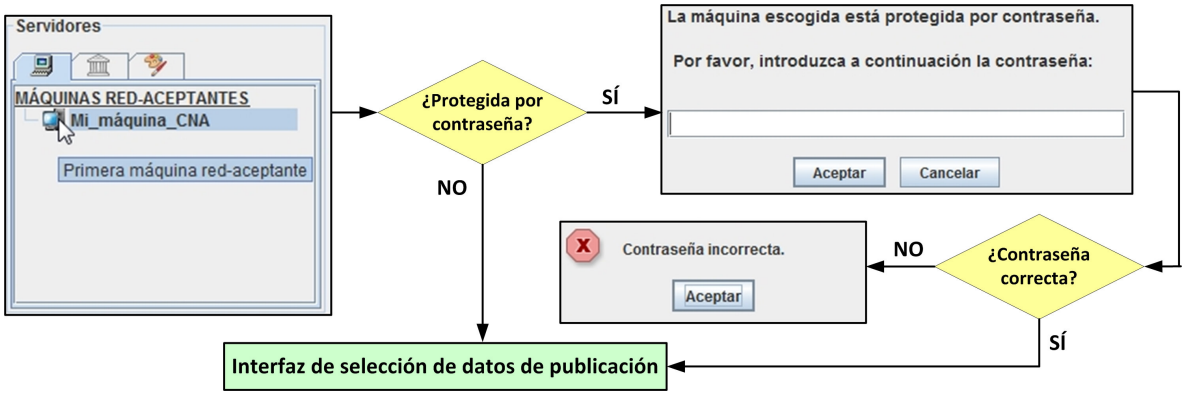

Figura 7.6: Interfaz de interacción con una máquina red-aceptante

Si la contraseña es correcta (o no la había), CAEAT muestra la interfaz de publicación de la agregación (o del servicio simple en caso de estar publicando únicamente un *bean*). A partir de este punto, el proceso de lanzamiento es idéntico, en lo que respecta a la interfaz de usuario, a la publicación del servicio desde la máquina local. El usuario deberá escoger los datos del servicio mediante las mismas interfaces que se presentaron en el capítulo 5.2.

La lógica interna de CAEAT se encarga de lanzar el servicio siguiendo el proceso descrito en el apartado 7.3. Al finalizar la operación, la instancia de CAEAT "lanzadora" del servicio resta como un cliente más del mismo, con derechos de posesión sobre ella (puesto que el usuario "lanzador" de un servicio equivale a su creador y publicador, pese a estar desplegando servicios en máquinas ajenas).

7.4.3. Gestión de CAEAT Network Acceptor

CNA no es un servicio "al uso" que pueda ser gestionable desde el exterior como cualquiera de los servicios o agregaciones de componentes publicados por la plataforma CAEAT. Por lo general, cuando una máquina se marca como red-aceptante se desea que lo sea permanentemente. La única operación de gestión posible sobre la máquina red-aceptante es su detención cuando se decida que ésta ha llegado al final de su vida.

La detención de una máquina red-aceptante es una operación de extremada delicadeza, puesto que implica también la detención de todos los servicios que pudiese estar manteniendo en el momento de la detención. Dado que por lo general cualquier usuario con permisos suficientes puede lanzar servicios a la máquina red-aceptante, el número de servicios gestionados por ésta será habitualmente alto. Por estas razones se ha prohibido la detención remota de un servicio CNA: para finalizar una ejecución de CNA se deberá realizar ésta desde la máquina física en la cual se está ejecutando.

La interfaz de *CAEAT Network Acceptor* cuando está gestionando algunos servicios es la siguiente:

Figura 7.7: Interfaz de CNA cuando está gestionando servicios lanzados desde el exterior

Nótese la similitud de esta interfaz con la del gestor de servicios publicados desde la máquina local, que fue presentado en el capítulo 4.5. Esta interfaz ofrece al usuario de la máquina red-aceptante información rápida sobre los servicios gestionados por dicha máquina que han sido lanzados desde los clientes remotos de CAEAT. Dado que el usuario de la máquina red-aceptante también ejerce como su administrador y supervisor, puede seleccionar un servicio y detenerlo a voluntad, sin importar quién fuese el propietario de éste. Esta funcionalidad es útil para corregir lanzamientos erróneos o inautorizados de agregaciones de servicios hacia la máquina red-aceptante.

El listado mostrado por la interfaz gráfica anterior es actualizado de manera dinámica cada vez que se recibe o se detiene un servicio desde el exterior. En el código 7.1 puede verse que uno de los métodos públicos del servicio CNA se denomina `actualizarInterfaz()`. Este método sirve para que los clientes de CAEAT puedan pedir a la máquina red-aceptante que actualice la interfaz

tras haber realizado cambios en su lista de servicios publicados. Estos cambios ejecutados desde el exterior por parte de CAEAT únicamente pueden ser:

- El lanzamiento de un nuevo servicio hacia la máquina red-aceptante
- La detención remota de un servicio que estaba siendo alojado por la máquina red-aceptante

En cualquiera de los dos casos, el usuario de la máquina red aceptante (en caso de tener abierta su interfaz) puede ver en tiempo real cómo se agregan o sustraen respectivamente servicios de la lista.

No obstante, tener abierta permanentemente la interfaz gráfica de CNA no será la manera más habitual de proceder. Como se ha dicho, se desea que CNA sea un módulo de software que corra permanentemente y en segundo plano en la máquina red-aceptante. Si dicha máquina es usada para otros fines (y se debe recordar que el entorno *Jini / Apache River* promueve la diversidad de roles de las entidades), tener una interfaz gráfica permanentemente abierta resulta incómodo para el usuario de la máquina. Se ha implementado a causa de ello un mecanismo de minimización al *SystemTray*.

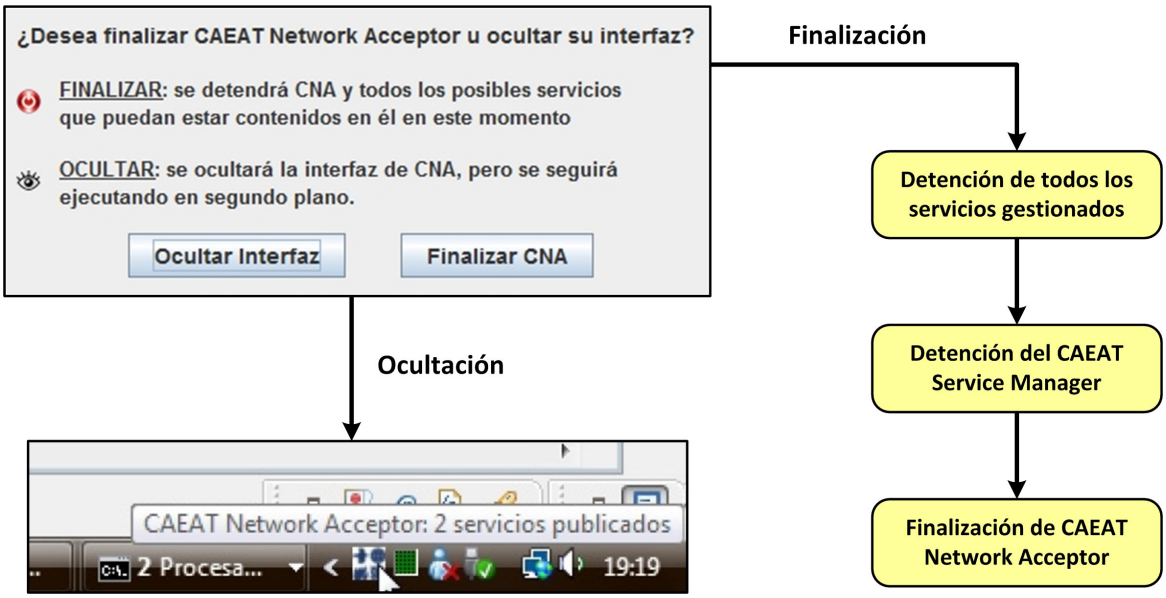

Figura 7.8: Uso de *SystemTray* para la ocultación de la interfaz de CNA

Cuando se intenta cerrar la interfaz gráfica de CNA o se pulsa sobre el botón “Salir”, se pregunta al usuario si desea la ocultación de la interfaz del programa o su finalización total. Si se desea su finalización, se detienen secuencialmente todos los servicios gestionados por la máquina red-aceptante (una barra de progreso informa del proceso), a continuación se detiene la instancia de *CAEAT Service Manager* que alojaba los servicios y el programa finaliza. En cambio, si se desea ocultar su interfaz, se minimiza CNA al *SystemTray*.

El *SystemTray* es una zona de la interfaz de usuario que implementan la mayoría de sistemas operativos en la actualidad en la que se visualizan en forma de icono los programas que se están ejecutando en segundo plano. En el sistema operativo *Windows*, por ejemplo, dicha zona es denominada “Área de notificación”. El lenguaje de programación *Java* incorpora, desde su versión 1.6, una clase denominada precisamente `SystemTray` que permite acceder a dicho espacio de los sistemas operativos de una manera independiente a la plataforma.

Haciendo un buen uso de la clase `SystemTray` es posible:

- Escoger el icono que se muestra en la zona del *SystemTray*. Este icono se denomina *TrayIcon*
- Generar un menú que se desplegará al realizar *clic* sobre el icono con el botón secundario del ratón (ver figura anterior)

- Elegir qué acción será llevada a cabo por defecto (de entre las que conforman el menú anterior) al realizar doble *clic* en el icono. En el caso de CNA, se restaura su interfaz.
- Asociar un *tooltip* o texto flotante al icono. En el caso de CNA, se muestra el número de servicios actualmente gestionados, información que se actualiza de manera dinámica a medida que los clientes lanzan y/o detienen servicios desde el exterior.

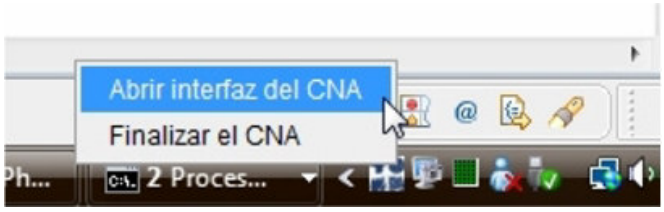

Figura 7.9: Menú contextual asociado al *TrayIcon* de *CAEAT Network Acceptor*

8. SERVICIOS ESPECIALES IMPLEMENTADOS

El presente proyecto ha acometido la adaptación al entorno remoto de la mayoría de servicios locales que se habían desarrollado para la plataforma CAEAT y que fueron presentados en el capítulo 1. Adicionalmente se ha desarrollado un meta-servicio como es *CAEAT Network Acceptor*: un servicio especial propio de CAEAT que ayuda al despliegue de otros servicios y que ha sido descrito en el capítulo anterior.

Este capítulo describe la motivación, arquitectura y funcionamiento de otros dos servicios especiales propios de CAEAT desarrollados durante la elaboración de este proyecto: los repositorios de librerías y los servidores de imágenes de tapiz. Estos nuevos servicios son visibles desde la interfaz de usuario de CAEAT a través del recuadro "Servidores", añadido debajo de las paletas de servicios y que comparten espacio con la visualización de las máquinas red-aceptantes disponibles, en una organización por pestañas fácilmente navegable.

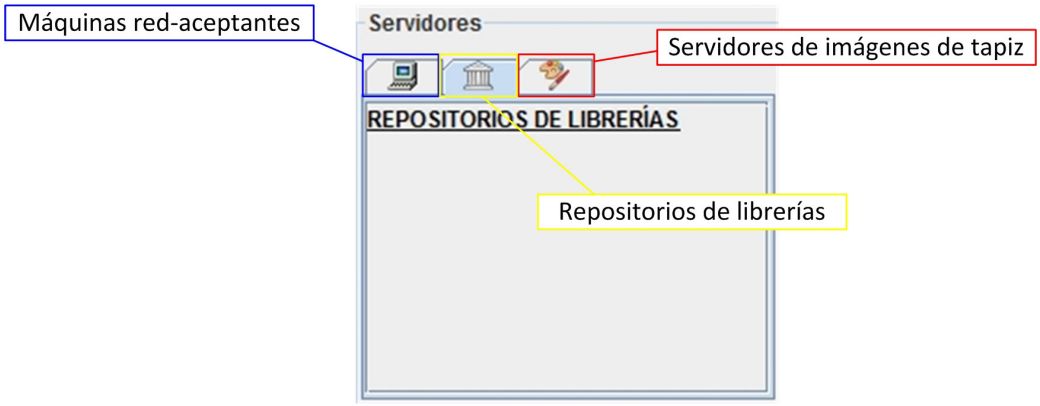

Figura 8.1: Visualización de la pestaña de repositorios de librerías

8.1. REPOSITORIOS DE LIBRERÍAS

En el contexto de CAEAT, se entiende por "librería" a un conjunto de *beans* aptos para ser utilizados con la herramienta de edición, que se encuentran encapsulados en un archivo de tipo *jar* y que guardan alguna relación temática y de utilidad entre ellos. En el capítulo 2 se presentaron las librerías desarrolladas para CAEAT: matemática, gráfica, de tratamiento de datos, etc.

Se detalló en aquel capítulo el procedimiento que se seguía para conseguir añadir librerías a la plataforma de manera dinámica en tiempo de ejecución: es posible seleccionar un archivo *jar* desde el sistema de ficheros de la máquina local para que CAEAT lo analice y añada los *beans* contenidos en él a la lista de componentes disponibles.

Con la adaptación a la lógica de procesado remoto de la plataforma, se desea conseguir también que la distribución y obtención de librerías no se limite únicamente al entorno local de la máquina que está ejecutando la plataforma CAEAT, sino que sea global y que por lo tanto dichas librerías puedan descargarse desde una serie de repositorios repartidos entre las entidades que forman una federación de servicios *Jini / Apache River*.

Con tal propósito se diseña el servicio de reposición de librerías, un servicio especial propio de CAEAT, que sigue la misma arquitectura que cualquier otro servicio de *Jini / Apache River*, y que permite el libre intercambio de librerías entre entidades pertenecientes a la red de servicios.

8.1.1. Requisitos a cumplir

Como paso previo a la definición de la arquitectura diseñada, se listan a continuación los requisitos y objetivos que se pretenden conseguir con la implementación del servicio de reposición de librerías:

- Los servidores de librerías darán a conocer su existencia mediante el registro de un objeto *Proxy* en los servidores de *Lookup*. Este *proxy* ofrecerá todos los métodos de comunicación necesarios entre cliente y servidor para conseguir el correcto intercambio de las librerías.
- La transmisión de los archivos *jar* que contienen las librerías de *beans* se realizará con la ayuda de un servidor FTP. Hasta el momento y para todos los servicios y procedimientos descritos, a la figura del servidor le bastaba con "compartir" sus recursos (los archivos *class* en el momento de la publicación, los iconos descriptivos de los servicios, etc.). Sin embargo, un repositorio de librerías debe ser capaz también de "aceptar" archivos procedentes de los clientes (los propios archivos *jar*), por lo que un servidor FTP resulta imprescindible.
- Se deberá poder inicializar un servidor de librerías desde la propia interfaz de CAEAT tal y como si se tratase de cualquier otro servicio (por ejemplo, una agregación). Deberá ser posible, opcionalmente, levantar un servidor de librerías protegido con contraseña para evitar que la totalidad de usuarios de CAEAT tengan acceso a las librerías almacenadas en él.
- CAEAT deberá ofrecer mecanismos para que el usuario pueda realizar las dos operaciones básicas sobre un servidor de librerías: la descarga y la subida de archivos *jar*. En descarga, será posible previsualizar el contenido de las librerías (los componentes que éstas contienen) antes de seleccionar y descargar una de ellas en concreto. En subida, se deberá seleccionar un archivo *jar* del entorno local de CAEAT (o de cualquier otra localización en la máquina local) para subir al servidor.
- Aunque se permitirá la libre circulación de archivos *jar* a lo largo de la red de servicios y entidades *Jini*, no se permitirá su libre utilización. Los archivos de las librerías se distribuirán cifrados mediante el algoritmo de encriptación AES, de tal manera que los clientes que obtengan una librería deberán disponer de la clave adecuada para poder descifrarla y hacer uso de ella.

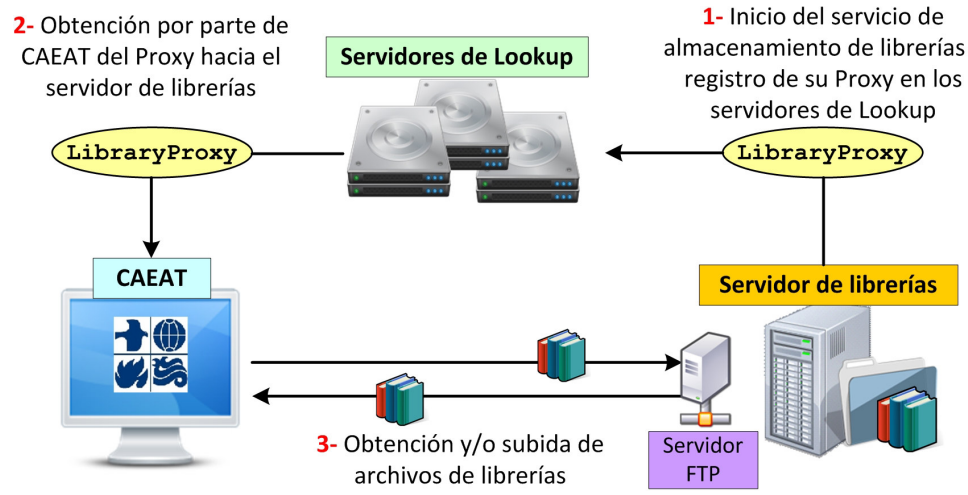

Figura 8.2: Arquitectura *Jini / Apache River* aplicada al diseño de servidores de librerías

8.1.2. Arquitectura diseñada

A continuación se describe la forma en que la arquitectura *Jini / Apache River* se ha utilizado para conseguir diseñar un servicio como el expuesto en el apartado anterior.

8.1.2.1. Diseño del servidor

Como en todo servicio *Jini*, el servidor consta de un objeto *Java* que actúa de *Server*, que se exporta para permitir acceso remoto y cuyo *Proxy* se registra en los servidores de *Lookup* para dar a conocer el servicio al resto de la red. En el objeto *Server* residen los atributos reales del proceso. Para los servidores de librerías estos atributos consisten en una serie de datos que ayudan a identificar el servidor y a mantener un catálogo de librerías que el usuario puede consultar como paso previo a la obtención de una librería. Entre estos datos se encuentran:

- Nombre y descripción del servidor, información que aparecerá en el lado del cliente como resultado de las búsquedas.
- Contraseña, si el servidor se ha inicializado bajo protección. En caso de haber sido así, el servidor no devolverá al cliente la información sobre el catálogo de librerías contenidas a menos que éste introduzca esta contraseña.
- URL de acceso a las librerías. La dirección FTP desde la cual el directorio que almacena las librerías del servidor es accesible "desde fuera" únicamente es comunicada al cliente en el momento de la descarga de la librería. Se debe recordar que, pese a que cliente y servidor hayan establecido una comunicación, el entorno de programación *Jini / Apache River* oculta mutuamente las direcciones IP de las entidades participantes, por lo que es una información que se debe obtener explícitamente si se desea disponer de ella a nivel programático.
- Obviamente, se debe disponer de un servidor FTP activo atendiendo a las peticiones que lleguen a la mencionada URL. Uno de los directorios de trabajo de la máquina del repositorio a los que deberá apuntar dicho servidor FTP será la carpeta en la que se guardan las librerías que se deseen servir.
- Catálogo de las librerías contenidas en el servidor. El catálogo almacena en primer lugar los nombres de las librerías. Para cada librería, se almacena junto a su nombre un mapa que asocia los nombres de todos los *beans* contenidos en la librería con una URL desde la cual el icono representativo de cada *bean* es accesible por entidades externas.

  La lista de iconos será usada en el lado del cliente para ofrecer una previsualización en formato agradable del contenido de cada librería. Al recibir una nueva librería, el servidor es el responsable de analizarla, copiar los iconos contenidos en ella a la carpeta pública de esa máquina y elaborar y añadir una nueva entrada de catálogo como las descritas anteriormente.

8.1.2.2. Servidor FTP utilizado

El diseño de los servidores de librerías se ha realizado dejando suficiente libertad al usuario para la elección del servidor FTP que considere más oportuno. Para el presente proyecto, y siguiendo la misma línea que en el caso de los servidores HTTP y de *Lookup*, se ha utilizado un servidor ligero, escrito en el lenguaje de programación *Java* y fácilmente configurable mediante la modificación de algunos parámetros en su fichero de configuración. El servidor elegido ha sido *Apache FTP Server*, cuya documentación oficial y última versión (1.0.6) pueden obtenerse en la referencia [**9**].

Pese a esta libertad de elección en el uso de una u otra herramienta, existen dos restricciones a tener en cuenta por el usuario que despliegue un servidor de librerías:

1. Se ha decidido reservar un espacio en la carpeta pública de CAEAT (aquélla en la que se copian los recursos públicos durante la publicación de un servicio) para el intercambio de archivos mediante FTP (el servidor de imágenes de tapiz, explicado en el apartado siguiente, también hace uso de éste). Será responsabilidad del usuario que despliegue el servidor FTP procurar que éste se encuentre sirviendo correctamente el directorio "CAEAT-Public/ftp".
2. Para mayor seguridad, se ha decidido no permitir la utilización como usuario anónimo del servidor FTP. El código fuente de CAEAT "ataca" el servidor FTP mediante la utilización de los siguientes datos de *login*:
   - **Nombre de usuario**: caeat
   - **Password**: S3N3T834NS_SA

   Es responsabilidad del usuario que despliega el servidor FTP la configuración del mismo para que reconozca estas credenciales o, si lo considera oportuno en su contexto, para permitir el uso anónimo del servidor FTP.

8.1.2.3. Diseño del cliente

El diseño de la lógica del cliente es mucho más sencillo que la del lado del servidor. Los *proxies* de los servidores de librerías implementan una interfaz del lenguaje de programación *Java* que es bien conocida por la plataforma CAEAT. Dicha interfaz se utiliza como plantilla a la hora de realizar un proceso de búsqueda o *discovery* de repositorios de librerías en todo el ámbito de la red.

Los servidores de librerías encontrados se listan en la pestaña correspondiente de la paleta de servicios especiales mostrada al inicio del presente capítulo. Los usuarios de CAEAT pueden seleccionar uno de ellos e interactuar con él mediante las interfaces y procedimientos que se presentarán en el apartado siguiente.

8.1.3. Ejemplo de aplicación

Este apartado ilustra los procesos de inicialización de un servidor de librerías y la interacción con él como entidad cliente en los dos sentidos (tanto subida como bajada). Se incluyen capturas de la interfaz de CAEAT para poner de manifiesto las nuevas herramientas y menús que se han implementado para poder interactuar de una manera sencilla con el nuevo servicio desarrollado.

8.1.3.1. Despliegue de un servidor de librerías

La inicialización de un servidor de librerías y el consecuente registro de su objeto *Proxy* en los servidores de *Lookup* se puede acometer desde la propia interfaz de CAEAT. Se ha proporcionado una entrada de menú para tal fin, en uno de los nuevos menús implementados en la interfaz de la plataforma.

Como se puede observar en la figura siguiente, se comprueba si en la máquina hay un servidor FTP activo como paso previo al lanzamiento de la interfaz de selección de datos. Esto se lleva a cabo realizando un intento de apertura de conexión FTP con la máquina local. Dado que el servidor FTP es un requisito imprescindible para el funcionamiento de los servidores de librerías, no se permite el despliegue de uno si no hay un servidor FTP presente.

Sin embargo, cabe destacar que sí se permite la activación de más de un repositorio de librerías en la misma máquina. Todos los repositorios arrancados en una misma máquina constituirán a todos los efectos servidores independientes. Así aparecerá en los resultados de las búsquedas de los clientes, que desconocerán que en realidad hay una única máquina tras todos ellos.

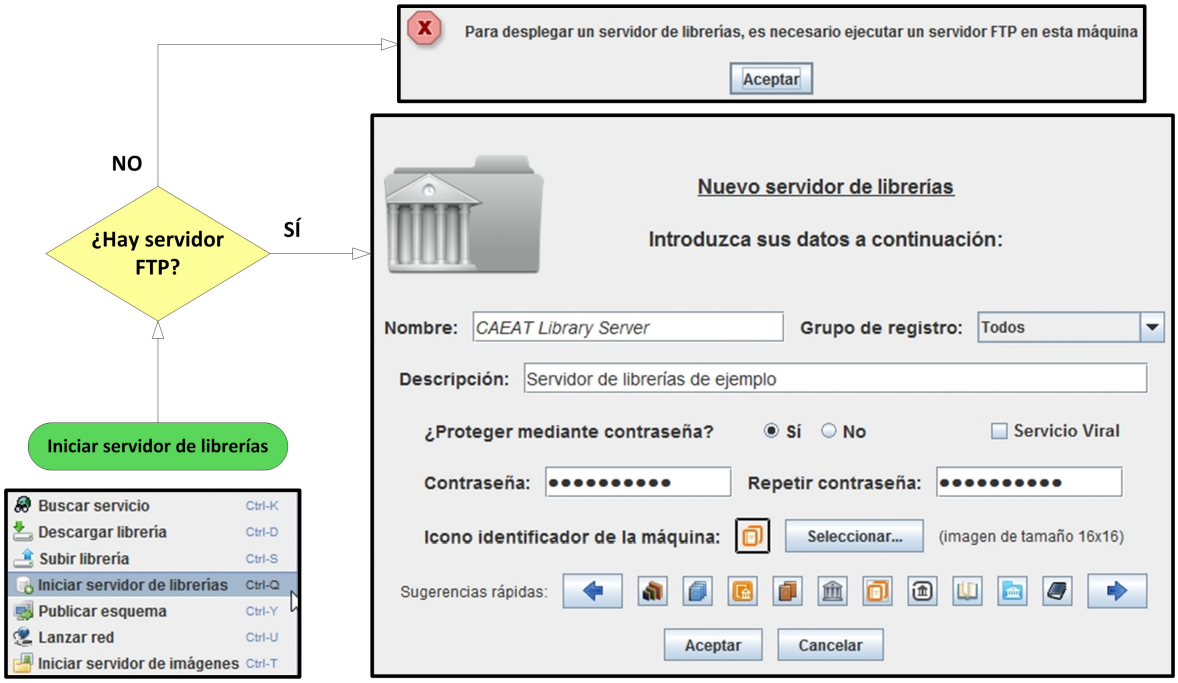

Figura 8.3: Interfaz de inicialización de un nuevo servidor de librerías

La interfaz de inicialización permite seleccionar todos los datos relevantes del servidor. Es posible elegir un nombre, una descripción y un icono identificador, datos que se mostrarán a los usuarios como resultados de las búsquedas. Es posible asimismo seleccionar el grupo de *Lookup* en el cual se registrará el *proxy* del servicio, aunque para este tipo de servicios especiales lo más habitual será buscar el máximo alcance de visibilidad. Se permite también determinar si el servicio será viral. Por último, es posible asociar una contraseña al servidor, para evitar que los usuarios que no dispongan de ella puedan navegar por su catálogo de librerías.

8.1.3.2. Subida de una librería

Una vez registrado el repositorio de librerías en los servidores de *Lookup*, los clientes pueden obtener su objeto *Proxy* al realizar una búsqueda de servicios. Como ya se ha explicado, los servicios de carácter especial se muestran en una paleta completamente aislada de la paleta de servicios locales y remotos. Esta paleta, del mismo modo que la otra, se actualiza automáticamente cada N minutos, donde N es un parámetro controlable por el usuario mediante el menú de opciones.

Realizando doble *clic* sobre las entradas de esta paleta es posible interactuar con el *Proxy* del servidor. Si éste se ha marcado como protegido por contraseña, se muestra un cuadro de diálogo que pide dicho dato al usuario. Cabe destacar que la contraseña auténtica no viaja por la red por motivos de seguridad: se envía la cadena introducida al usuario al servidor y es éste quien la computa y comunica si es correcta mediante el retorno de un valor de tipo *boolean* (*true* / *false*).

Si la contraseña es correcta (o si no había tal contraseña), se pregunta al usuario qué acción desea realizar con el repositorio: subir una librería o descargarla. Para el caso de la subida, se abre un cuadro de selección de archivos para elegir la librería a subir. Aunque por defecto este cuadro muestra el directorio en el que se guardan las librerías que CAEAT está usando en ese momento, es posible seleccionar cualquier archivo *jar* del sistema de ficheros de la máquina local. Las interfaces de usuario con las que se puede llevar a cabo todo lo explicado anteriormente se muestran en la figura siguiente:

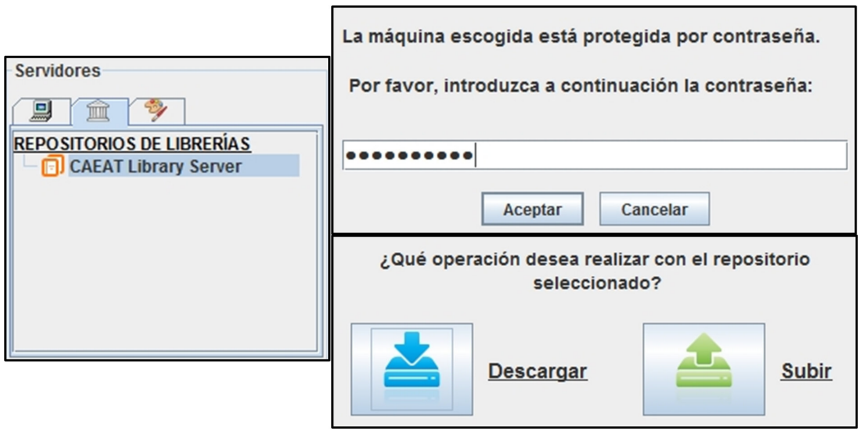

Figura 8.4: Interfaces de interacción con un repositorio de librerías

Una vez seleccionado un fichero, se solicita al servidor la dirección FTP remota a la que se debe subir el archivo, se abre una conexión con dicha dirección y se realiza la transferencia. En recepción, el servidor incorporará los datos de la librería a su catálogo interno para mostrarla a los usuarios junto con las demás (nombre, *beans*, iconos, etc.)

8.1.3.3. Descarga de una librería

El proceso de descarga de una librería es análogo al de subida pero introduce una interfaz de usuario nueva como paso previo a la descarga: la ventana que permite navegar por el catálogo de librerías del servidor y previsualizar los componentes contenidos en cada una de ellas.

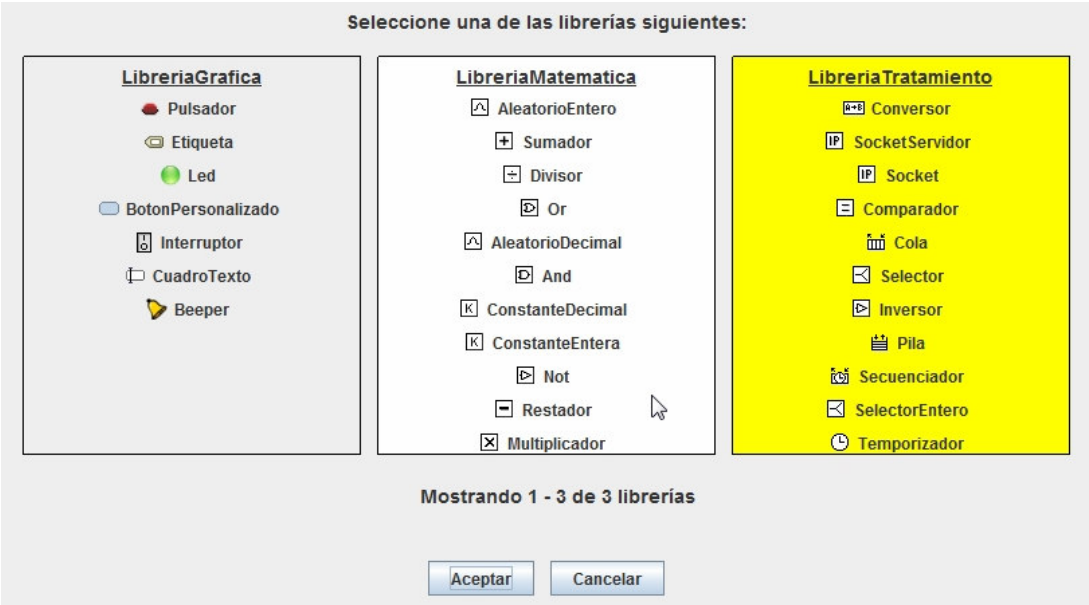

Figura 8.5: Interfaz de navegación y visualización del catálogo de librerías de un repositorio

La interfaz de navegación de librerías muestra de manera agradable al usuario el contenido de éstas. Si hay multitud de ellas, se ofrecen botones para navegarlas rotativamente. Para seleccionar una basta con realizar un *clic* sobre el recuadro que la contiene y éste quedará marcado en color amarillo.

Una vez seleccionada una librería, el cliente pide al servidor la dirección FTP que se debe acometer para tener acceso a dicha librería. Se solicita la transferencia de la librería seleccionada, copiándose el archivo *jar* de la misma al espacio local de CAEAT. Si la librería no se encuentra encriptada (ver apartado siguiente), ésta es inmediatamente analizada por la plataforma y sus *beans* son añadidos a la paleta de componentes locales para su uso normal.

8.1.4. Cifrado de librerías

El esquema de distribución diseñado permite el libre intercambio de los archivos de librerías a lo largo de toda la federación de entidades *Jini* sin ningún tipo de limitación. Si bien es cierto que se han tomado ciertas medidas (protección opcional de los repositorios con una contraseña, exclusión del usuario anónimo del servidor de FTP, etc.), los archivos *jar* que contienen la librería se distribuyen tal y como son generados.

En el capítulo 2 se expusieron los casos de uso del entorno CAEAT / *SeNetComponents*, y se presentó la figura del programador de librerías. El cometido de esa persona o entidad será la de programar librerías de componentes compatibles con CAEAT, actualizar las ya existentes, desarrollar librerías compatibles con otras plataformas de edición similares, etc. Los repositorios de librerías constituyen un canal de distribución del producto tremendamente potente para este "actor".

Sin embargo, es necesaria la introducción del concepto de "licencia" en el marco de la generación y distribución de librerías. Los desarrolladores de librerías necesitarán algún mecanismo de protección de su negocio que permita al mismo tiempo características que a priori parecen antagonistas: conseguir la máxima rapidez y distribución de las librerías pero al mismo tiempo garantizar que únicamente los clientes autorizados pueden hacer uso de ellas. Se ha diseñado para tal fin un esquema de cifrado de las librerías que hace uso del algoritmo de encriptación AES.

8.1.4.1. Esquema de cifrado

El lenguaje de programación *Java* ofrece en su paquete de criptografía *javax.crypto* clases para la realización de operaciones de cifrado mediante el uso de multitud de algoritmos. Para el cifrado de las librerías del presente proyecto se ha escogido el algoritmo AES (*Advanced Encryption Standard*) por ser el algoritmo de encriptación simétrica más usado en la actualidad, resultando más robusto que su predecesor DES. Además, no requiere grandes capacidades en términos de memoria o capacidad de computación para ser implementado.

AES admite encriptación mediante el uso de claves de 128 bits y 256 bits de longitud. Para el presente proyecto se han utilizado claves de 128 bits buscando la máxima compatibilidad, ya que algunas versiones antiguas de la Máquina Virtual de *Java* no soportan una longitud de clave de 256 bits para el algoritmo AES. La máxima longitud de clave también presenta problemas en algunas implementaciones de la JVM para las distintas versiones del sistema operativo *Android*.

Para realizar un cifrado mediante AES es necesaria la generación de una clave secreta (clase `SecretKey` del lenguaje *Java*) de la longitud anteriormente citada. Esta clave puede ser generada mediante dos procedimientos:

- Generación totalmente aleatoria por parte de la JVM. En este caso, para poder descifrar en recepción, sería necesario guardar la `SecretKey`, generalmente como una sucesión totalmente aleatoria de caracteres en formato textual con una codificación determinada (que se deberá conocer). Es un esquema poco manejable en un ambiente distribuido como CAEAT / *SeNetComponents*.
- Generación a partir de algún tipo de información secreta memorizable por humanos (*password*). En este caso es necesario también el uso de un vector de *bytes* que sirva como semilla para iniciar el generador aleatorio de la `SecretKey`. Este vector de *bytes* es de dominio público, y se debe comunicar al receptor. Si éste además conoce el *password* (la información secreta), puede generar una `SecretKey` idéntica a la utilizada en emisión, y por lo tanto descifrar la información. Es el esquema adoptado en el presente proyecto.

Pese a que la primera noción al pensar en el mecanismo de cifrado lleva al cifrado de archivos, el lenguaje de programación *Java* es más flexible y permite cifrar flujos de *bytes* o *streams*: aquellos

objetos que representan fuentes de información a través de las cuales leer de un fichero, del teclado, de un puerto, de un *buffer* en memoria, etc. Es posible, por lo tanto, cifrar y descifrar cualquier ristra de *bytes* en memoria, sin que ésta represente necesariamente un archivo en el sistema de ficheros de la máquina.

La característica anterior aplicada a las librerías de CAEAT permite diseñar un esquema de cifrado muy robusto en el cual los objetos cifrados no son los archivos *jar* de las librerías, sino los archivos binarios de Java (archivos *class*) contenidos en él. En recepción, y en el momento de cargar una librería, se descifrarán estos archivos únicamente en memoria y en tiempo de ejecución, no dejando rastro en el sistema de ficheros de la máquina de los archivos binarios en claro, únicamente de los cifrados. El resto de recursos contenidos en el archivo *jar* de la librería (imágenes, iconos, archivo *Manifest.mf*, etc.) permanecen inalterados.

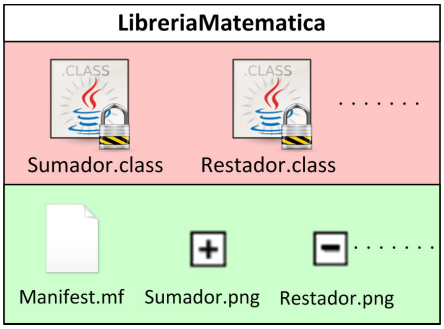

Figura 8.6: Contenido de una librería cifrada. Algunos recursos (en verde) no son cifrados

8.1.4.2. CAEAT Library Encrypter

El desarrollador de la librería, como su propietario y creador, es la figura que debe tener la potestad de decidir si ésta debe distribuirse cifrada o en claro. En caso de cifrarla, dicha operación se debe poder llevar a cabo sin la ejecución de la herramienta CAEAT, puesto que el usuario de dicha herramienta no necesita funciones de cifrado. También se puede dar la circunstancia de que el desarrollador de la librería no disponga de la herramienta CAEAT (ver el diagrama de casos de uso del capítulo 2.10). Es por ello que la operación de cifrado de librerías se lleva a cabo en un programa autónomo e independiente a CAEAT que se ha denominado *CAEAT Library Encrypter* (CLE), cuya interfaz de usuario se muestra a continuación:

Figura 8.7: Interfaz de usuario de *CAEAT Library Encrypter*

Desde la interfaz se permite escoger el archivo *jar* de origen y su ruta destino de salida. Se debe especificar la contraseña a partir de la cual se generará la `SecretKey` necesaria para el cifrado. El procedimiento seguido para el cifrado de una librería es el siguiente:

1. Se extrae completamente todo el contenido del archivo *jar* a una carpeta de trabajo temporal que será eliminada al final del proceso.
2. Se genera una `SecretKey` de 128 bits para su uso con AES a partir de un vector de *bytes* elegido al azar y de la contraseña escogida por el usuario.
3. Se agrega al archivo *Manifest.mf* de la librería un nuevo atributo: la representación textual del *array* de *bytes* (ya que es la información pública de la `SecretKey`).
4. Se itera sobre todos los archivos de la librería y se cifran únicamente los archivos de código binario (de tipo *class*), dejando el resto de los recursos inalterados.
5. Se vuelve a generar el archivo *jar*, utilizando la versión cifrada de los archivos *class* y la nueva versión de *Manifest.mf*. Se añade al nombre de archivo el prefijo "[C]" para evitar sobrescrituras en caso de que las localizaciones de origen y destino sean las mismas.

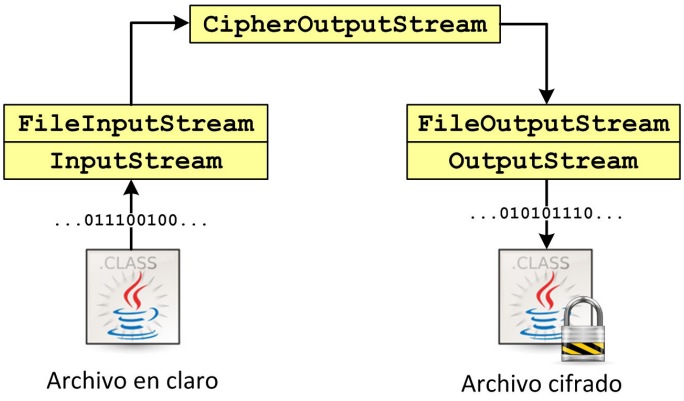

Figura 8.8: Clases *Java* implicadas en el cifrado de un archivo

El diagrama anterior muestra la estructura de clases del lenguaje de programación *Java* implicadas en el cifrado de un archivo. La clase `CipherOutputStream` encapsula de manera transparente al resto de la aplicación los detalles del cifrado. Las herramientas de cifrado desarrolladas hacen uso de esta clase para inicializarla al algoritmo adecuado (AES en este caso) y asociarle un objeto cifrador que genera la `SecretKey` apropiada.

8.1.4.3. Descifrado en tiempo de ejecución

La razón por la cual se han cifrado únicamente los archivos *class* binarios de las librerías es la de permitir el descifrado en memoria y en tiempo de ejecución, evitando dejar rastro en el sistema de ficheros de los archivos descifrados. Para ello se ha aprovechado el hecho de que las clases contenidas en los archivos *jar* de las librerías son cargadas con un cargador de clases customizado.

En el capítulo 4.4 se detallaron los diferentes cargadores de clases que son usados en la plataforma CAEAT. En el lenguaje de programación *Java* es posible sobrescribir los métodos que se encargan de la localización, resolución y carga de clases para customizar estas operaciones a las necesidades del programador. En el presente proyecto, *CargadorClasesJar* es un cargador que se encarga de localizar y cargar las clases contenidas en los archivos *jar* de las librerías.

Para cargar una clase, todo cargador de clases debe ser capaz en algún momento de convertir un conjunto de *bytes* en una definición de clase en la memoria de la JVM, lista para ser instanciada y utilizada. Nótese que la definición de una clase se realiza de manera genérica en base a un conjunto de *bytes* y no a un archivo *class* para poder facilitar la carga de clases procedentes de localizaciones

remotas. Para realizar la definición, se utiliza el método *defineClass*, implementado por defecto por todos los cargadores de clases. Este método es llamado por *loadClass* y *findClass*, los métodos básicos de carga de clases.

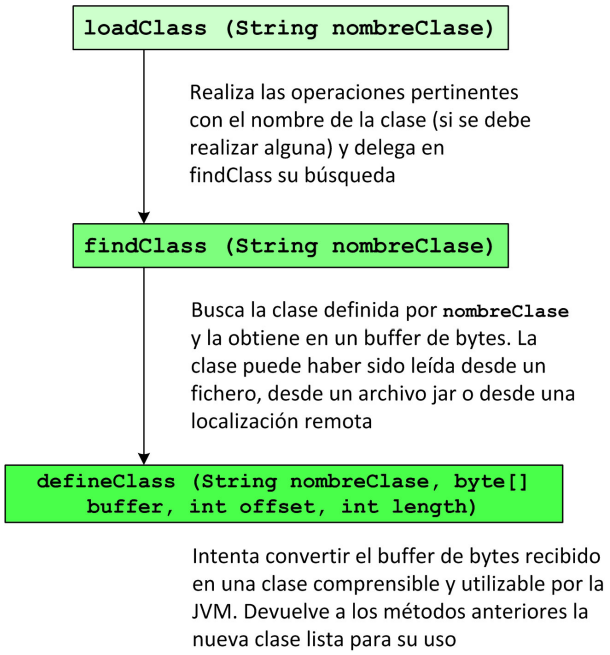

Figura 8.9: Mecanismo de resolución y carga de clases del lenguaje de programación Java

El *ClassLoader* customizado escrito para CAEAT puede realizar cuantas operaciones sean necesarias antes de la llamada a *defineClass*. Esto significa que, una vez se tiene el archivo de la clase almacenado en memoria como una ristra de *bytes*, es posible actuar sobre ella antes de entregarla a *defineClass* para que la resuelva como una nueva clase. En este punto es posible, pues, descifrar la clase en tiempo de ejecución si ésta se encuentra cifrada.

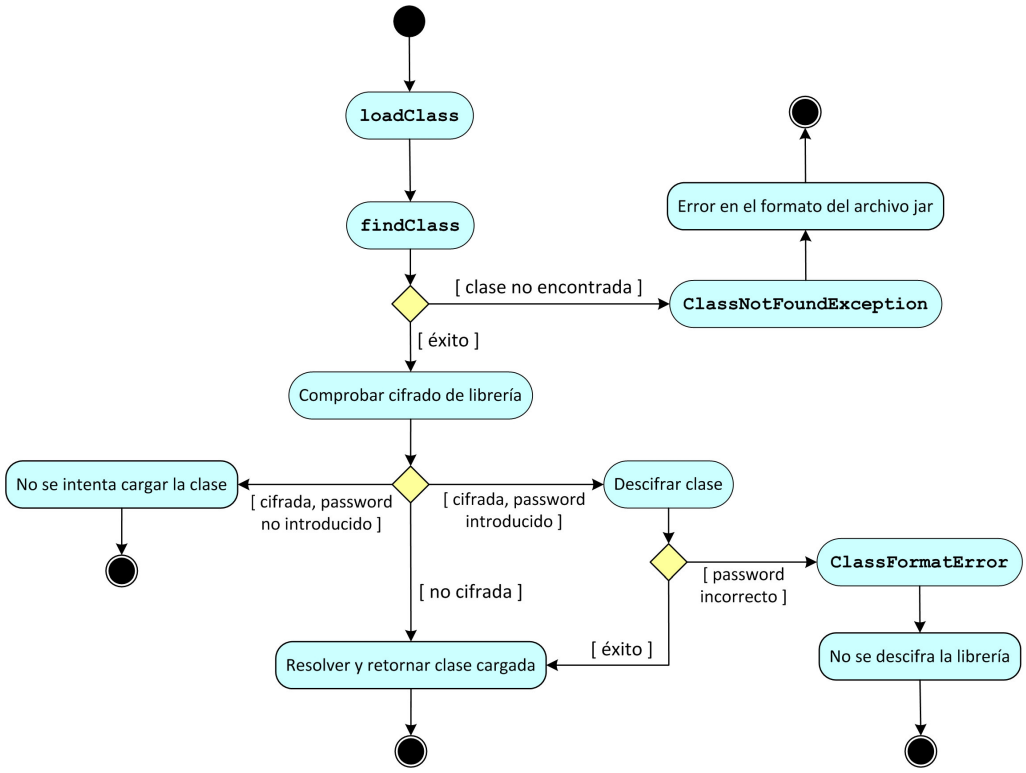

Figura 8.10: Diagrama de secuencia del descifrado dinámico de las clases de CAEAT

Cada librería insertada en CAEAT tiene un *CargadorClasesJar* asociado a ella, y cada uno de estos objetos encapsula un cifrador AES con clave de 128 bits. Este cifrador se inicializará únicamente si se detecta que la librería se encuentra cifrada: en caso contrario el funcionamiento de *CargadorClasesJar* será el habitual que ya se detalló en el capítulo 4.4. Si la librería se encuentra cifrada y su *password* ha sido introducido el cifrador será convenientemente inicializado, y será utilizado para descifrar el *buffer* de *bytes* que representa la clase antes de entregarlo a la Máquina Virtual de Java mediante *defineClass*.

Los *bytes* descifrados entregados a la JVM no llegan a tener presencia en el sistema de archivos de la máquina, y únicamente existen en memoria durante el tiempo de vida de la aplicación. Existe una única y lógica excepción: si el usuario decide publicar servicios provinentes de una librería cifrada, sus clases públicas (*Proxy*, *Wrapper*, etc.) deben ser copiadas en el directorio público del servidor HTTP en claro, de manera que sean comprensibles por los clientes del servicio.

8.1.4.4. Ejemplo de utilización

Se presenta a continuación un ejemplo de manejo, desde la óptica de CAEAT, de librerías cifradas. En caso de inserción de una librería cifrada, el *ClassLoader* customizado fracasará en la carga de los archivos *class*, puesto que al encontrarse cifrados resultarán incomprensibles para la Máquina Virtual de *Java* (no podrán ser resueltos). En este caso se captura la excepción lanzada y no se procede a la carga automática de la librería: únicamente se inserta en la paleta una referencia a la misma con un icono específico que indica el estado de encriptación de la misma.

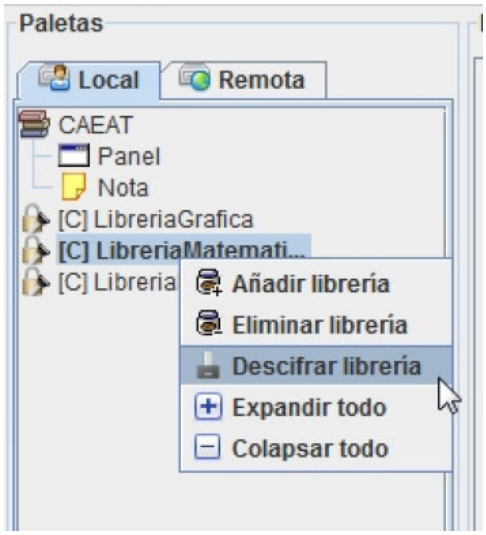

Figura 8.11: Representación en la paleta local de librerías encriptadas

Como se puede ver en la figura anterior, la librería resta cifrada en el espacio local de CAEAT a la espera de que el usuario inserte una contraseña. Esta operación se puede realizar mediante el menú contextual de los elementos de la paleta, o mediante un intento de despliegue, realizando doble *clic*, sobre la entrada que representa a la librería en la paleta. En cualquier caso, un cuadro de diálogo pide al usuario la inserción de la contraseña.

Una vez insertada la contraseña, los cifradores encapsulados por los *ClassLoaders* se encargarán de descifrar los archivos *class* antes de entregarlos al método *defineClass*. Si la contraseña era incorrecta, la resolución de las clases volverá a fracasar y la situación será la misma que al inicio. Si la contraseña era correcta, todas las clases cargadas desde ese archivo *jar* pasarán a partir de ese momento a través del cifrador, y la librería será automáticamente extraída tal y como se realizaba hasta el momento.

Se debe destacar que, una vez descifrada una librería, la intervención del cifrador encapsulado en el *ClassLoader* de esa librería es totalmente transparente a la plataforma CAEAT: el resto de la lógica es idéntica al caso de los archivos *jar* descifrados. Nótese que se cumple la premisa de partida del diseño del presente esquema de cifrado: las clases descifradas únicamente existen en memoria y en tiempo de ejecución, y desaparecerán cuando se cierre la instancia de CAEAT en curso. Si el usuario decide subir una librería a un repositorio, se subirá el archivo *jar* que reside en el sistema de ficheros, que contiene los archivos *class* cifrados.

8.2. SERVIDORES DE IMÁGENES DE TAPIZ

Una de las mejoras introducidas en la plataforma de edición CAEAT durante la elaboración del presente proyecto que no guarda relación con el objetivo principal del mismo consiste en la posibilidad de asociar imágenes a los componentes del tapiz para que la elaboración de agregaciones resulte más intuitiva y visual. Esta mejora se encuentra explicada con más detalle en el anexo D.3.

La posibilidad de asociar imágenes a los componentes del tapiz lleva al diseño de entidades que proporcionen acceso a dichas imágenes. De la misma manera que se ha realizado en el caso de las librerías, los clientes serán capaces de sondear la red en busca de imágenes en caso de no disponer de ellas en el entorno local (o de no disponer de la imagen deseada). Este apartado describe la arquitectura implementada para los servidores de imágenes y un ejemplo de inicialización y uso de un servidor.

8.1.1. Requisitos a cumplir

Los requisitos de implementación y comportamiento deseados para los servidores de imágenes de tapiz se pueden resumir en los puntos siguientes:

- Los servidores deben proporcionar un *Proxy* capaz de ser registrado en los servidores de *Lookup* para que los clientes puedan acceder a él. El *Proxy* ofrecerá todos los métodos necesarios para garantizar la correcta transmisión de imágenes entre servidor y clientes.
- Será posible inicializar servidores de imágenes desde la propia interfaz de la plataforma CAEAT, de una manera similar a la que se implementó para los servidores de librerías.
- La distribución de imágenes será totalmente libre (en este caso, no se protegerán los servidores con contraseña ni se cifrarán los archivos).
- Los servidores de imágenes serán bidireccionales: podrán ofrecer imágenes a los clientes y también aceptar las que éstos les envíen. Sin embargo, la función principal de los servidores será la descarga de imágenes, por lo cual dicha descarga se canalizará a través del servidor HTTP, siendo el servidor FTP optativo (pero recomendable si se quiere proporcionar también la función de subida de imágenes).
- Los clientes, si lo desean, podrán interactuar con los servidores individualmente para bajar o subir una imagen. Sin embargo, CAEAT también debe ofrecer una interfaz para gestionar la búsqueda de todos los servidores de imágenes disponibles y listar las imágenes contenidas en todos ellos, para que el usuario pueda elegir la que desee haciendo abstracción del número de servidores reales que realmente hay detrás del catálogo de imágenes mostrado.
- Los servidores de imágenes dispondrán de un directorio en el cual el usuario que despliega un servidor de imágenes deberá copiar los archivos correspondientes. El catálogo del servidor se generará automáticamente mediante análisis del contenido de dicho directorio.

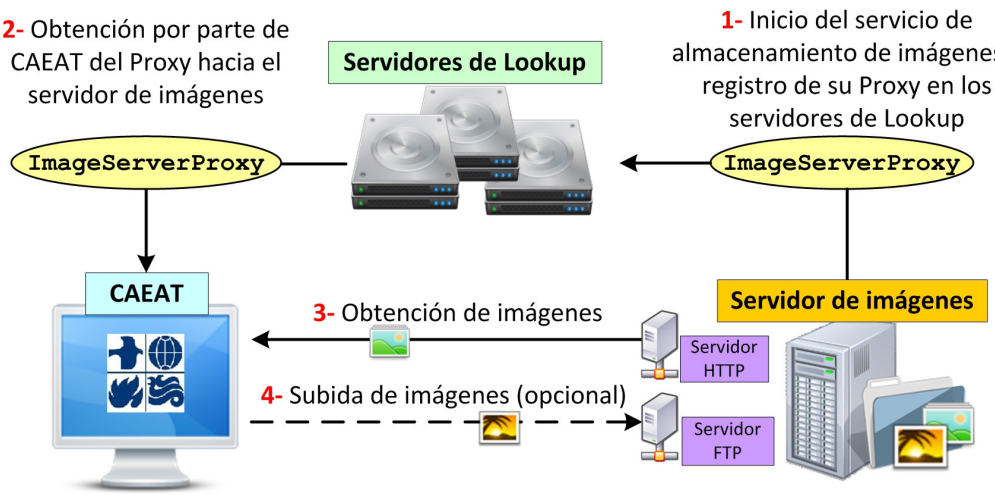

Figura 8.12: Arquitectura *Jini / Apache River* aplicada al diseño de servidores de imágenes

8.1.2. Arquitectura diseñada

Se describen a continuación los procedimientos y estrategias adoptados para utilizar la arquitectura *Jini / Apache River* para el diseño de los servidores de imágenes.

8.1.2.1. Diseño del servidor

Del mismo modo que en el caso de los servidores de librerías, los servidores de imágenes disponen de un objeto *Server* en el cual residen los atributos reales del proceso. Dicho objeto genera, en el momento de la exportación, un objeto *Proxy* que se registra en los servidores de *Lookup* para dar a conocer el servicio a los clientes. Los atributos que residen en el *Server* y que definen al servidor son los siguientes:

- Nombre y descripción del servidor, datos que aparecerán en el lado del cliente como resultado de las búsquedas.
- Catálogo, consistente en un mapa que asocia los nombres de las imágenes con la URL desde la cual éstas son visibles “desde el exterior”. Para bajada de imágenes se aprovecha el servidor HTTP, por lo que estas URL siguen dicho protocolo. El catálogo de imágenes se refresca cada vez que un cliente lo solicita, mediante el análisis del contenido de la carpeta de imágenes y la generación de las URL’s que apuntan a cada imagen válida. Cabe destacar que ante el encuentro de imágenes no válidas (ya sea por tamaño o formato) o de archivos que no son imágenes, ésta/os son ignorada/os.
- Ruta hacia la carpeta del sistema de ficheros en el cual se almacenan las imágenes. Por comodidad, se ha decidido que esta carpeta esté contenida dentro de la carpeta pública “CAEAT-Public” en la que se copian los recursos públicos de los servicios, bajo la ruta “CAEAT-Public/ftp/imagenesBeans”.
- Dirección FTP a través de la cual la carpeta anterior es accesible desde el exterior. El uso de esta característica es opcional y únicamente se dará en caso de que el usuario que despliega un servidor de imágenes desee permitir que los usuarios puedan subir imágenes a él, además de obtenerlas desde él. La dirección FTP almacenada ya tiene en cuenta la decisión de no permitir el uso anónimo de los servidores FTP empleados en CAEAT, por lo que encapsula el nombre de *login* y la contraseña correspondiente.

8.1.2.2. Manejo de imágenes a través de la red

Ya se ha comentado que la función principal de los servidores de imágenes será la descarga de imágenes por parte de los clientes, proceso que se gestiona aprovechando el servidor HTTP de *Apache River* que necesariamente se encontrará activo en la máquina del servidor. Sin embargo, se ha decidido que opcionalmente los clientes también puedan tener oportunidad de subir las imágenes desde su entorno local al servidor.

Para tal fin se reaprovecha el servidor FTP utilizado para el intercambio de librerías, con la misma configuración. Si el usuario que despliega el servidor de imágenes desea que éste sea apto para la recepción de imágenes, es su responsabilidad garantizar que exista un servidor FTP en la máquina local configurado tal y como se detalló en el capítulo dedicado al intercambio de librerías. Si un cliente desea subir una imagen a un servidor cuya máquina no está ejecutando un servidor FTP, se le informa de que tal operación no es posible.

8.1.2.3. Diseño del cliente

Por lo general, los clientes (usuarios de CAEAT) acudirán a los servidores de librerías cuando no encuentren imágenes en su espacio local (o no dispongan de una imagen apropiada). El comportamiento por defecto es el de buscar los *proxies* de todos los servidores de imágenes presentes en la red. Esta operación es posible porque CAEAT conoce la interfaz que implementan los servidores de imágenes y es capaz de realizar una búsqueda únicamente de este tipo de servicios. Con los catálogos de imágenes de todos los *proxies* en su poder, CAEAT muestra la interfaz de selección de imagen (ver apartado siguiente), con lo que oculta al cliente el número real de servidores que se han encontrado.

Sin embargo, también es posible interactuar con un servidor individualmente, pues éstos se muestran en el recuadro de servidores de la paleta. En este caso se muestra únicamente el catálogo de ese servidor. Sea cual sea el método, al seleccionar una imagen ésta es copiada al entrono local del CAEAT (el cliente dispondrá de ella en adelante) y se asocia al componente del tapiz con el que se estuviese trabajando en ese momento.

8.1.3. Ejemplo de aplicación

Se muestran a continuación los procesos de inicialización de un servidor de imágenes y su utilización por parte de los clientes, incluyendo capturas de la interfaces gráficas que se han diseñado para CAEAT con el fin de facilitar estas tareas al usuario.

8.1.3.1. Inicialización del servidor

La interfaz de usuario de inicialización de servidores de imágenes es muy similar a la de inicialización de repositorios de librerías. Se permite seleccionar datos básicos sobre el servicio, como su nombre, su descripción, el grupo de registro en el que será visible, y si éste será viral o no. Se informa al usuario que está desplegando el servicio del directorio en el cual deberá copiar las imágenes que desea compartir a través del servidor. Este directorio no es seleccionable ya que se debe encontrar dentro de la estructura creada automáticamente para compartir los recursos públicos de CAEAT. Se ofrece un botón para abrir dicho directorio y empezar a copiar las imágenes deseadas. Opcionalmente se permite asociar un icono descriptivo al servicio.

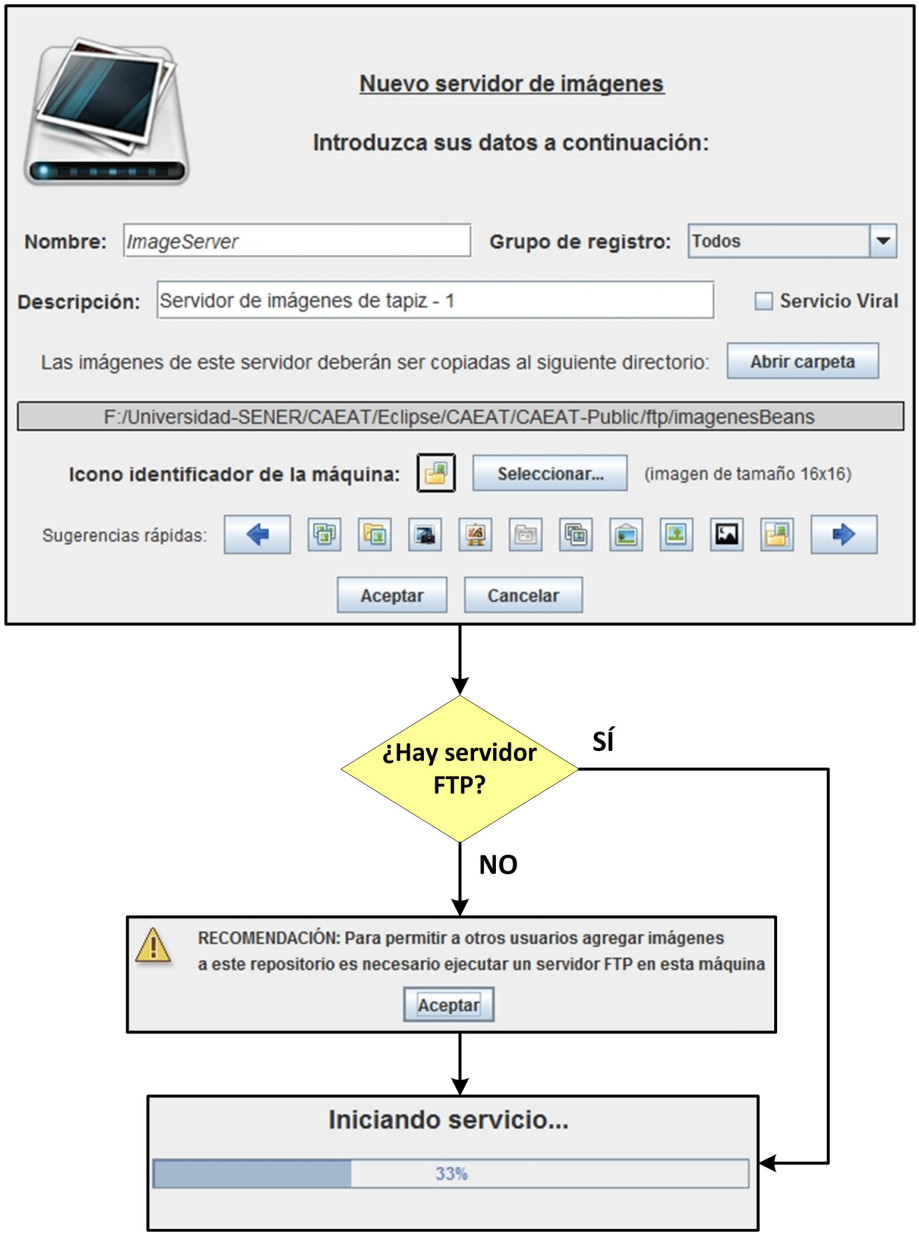

Figura 8.13 Interfaz y lógica de inicialización de servidores de imágenes

Tras la selección de datos se procede a la publicación del *proxy* del servicio en los servidores LUS adecuados y a la inicialización del servicio. Si se detecta que no hay un servidor FTP funcionando en la máquina local, se informa al usuario de que es recomendable iniciar uno si desea que los clientes sean capaces de subir imágenes al servidor de imágenes.

8.1.3.2. Búsqueda de imágenes

Desde el lado de los clientes, la forma más habitual de interacción con los servidores de imágenes será la de solicitar una búsqueda completa en toda la red desde la interfaz de selección de imágenes del menú contextual de un componente. La solicitud puede venir provocada por el hecho de no disponer de imágenes en el entorno local o por no disponer de una imagen adecuada a lo que el usuario está buscando. En cualquier caso, se realiza una búsqueda de todos los servidores de imágenes de la red y se lista su contenido, haciendo abstracción del número de servidores encontrados realmente, de su redundancia y de su ubicación.

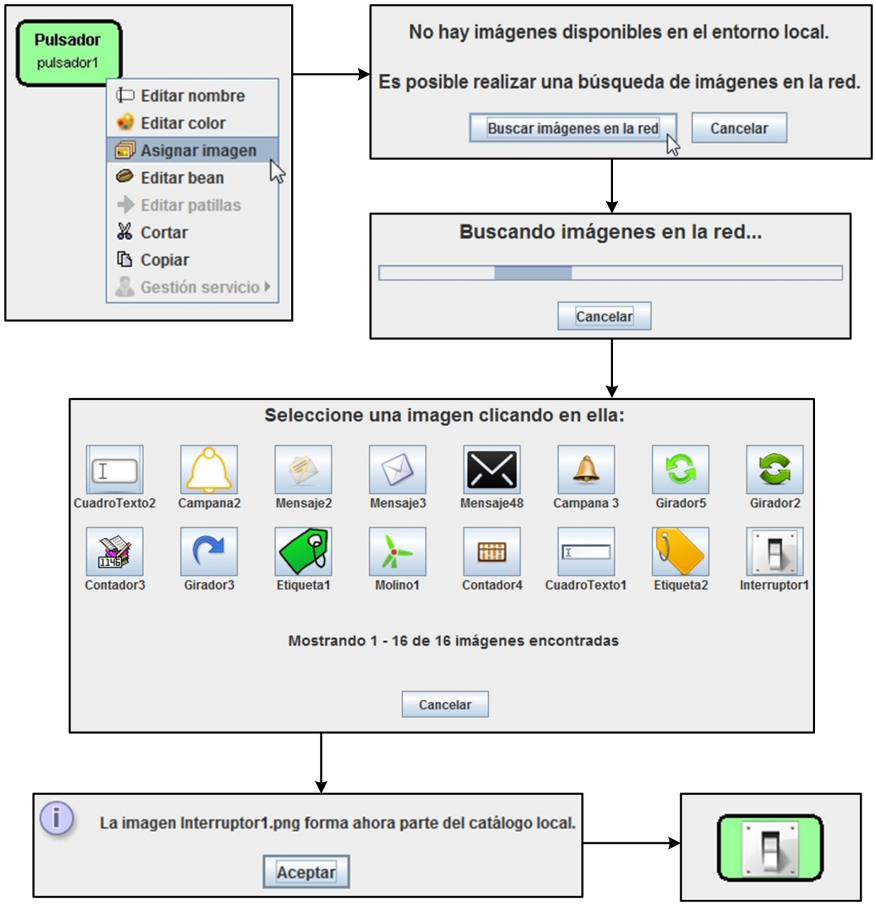

Figura 8.14: Interfaces gráficas de búsqueda de imágenes por la red

Para mayor comodidad del usuario, las imágenes son encapsuladas en botones, por lo que basta con realizar clic sobre alguna de ellas para escogerla. La interfaz se adapta al número de imágenes encontradas, mostrando un máximo de 48 imágenes al mismo tiempo y ofreciendo botones para rotarlas a izquierda y derecha en caso de necesitar más páginas. La imagen elegida se copia permanentemente en el entorno local de CAEAT para futuros usos, y al mismo tiempo se asocia al componente sobre el cual se estuviese llevando a cabo la edición.

Aunque menos habitual y menos práctico, es posible navegar por el contenido de un único servidor si se interactúa con él mediante su representación en la paleta de servidores de CAEAT. Dado que en este caso no habrá ningún componente del tapiz seleccionado, la imagen obtenida únicamente se copiará en el entorno local.

8.1.3.3. Subida de una imagen

Por lo general, es responsabilidad del usuario que inicializa el servidor de imágenes el correcto aprovisionamiento de archivos del mismo. Sin embargo, se ha querido dar al usuario la posibilidad de subir al servidor las imágenes que desee. Para ello, como ya se ha comentado, es necesario que exista un servidor FTP corriendo en la máquina en la que el servidor de imágenes ha sido inicializado.

Para subir una imagen a un servidor, es necesario realizar doble *clic* sobre su entrada en la paleta de servidores de CAEAT y seleccionar la opción de subir imagen. Se abrirá a continuación la interfaz de selección de imágenes alojadas en el entrono local. Lógicamente, si no se dispone de imágenes aptas en el entorno local no será posible subir ninguna. Tras seleccionar una imagen, se pide al servidor la dirección FTP que se debe “atacar” y se trata de realizar la transferencia del archivo. En caso de no encontrarse disponible un servidor FTP en el lado del servidor, se informa al usuario en este punto.

9. CONCLUSIONES Y LÍNEAS FUTURAS

Se presentan en este capítulo los resultados más importantes obtenidos durante la elaboración del presente proyecto; así como los objetivos cumplidos, enumerando también los hitos secundarios que no figuraban como objetivos principales al principio del mismo. Se enumeran asimismo las líneas de trabajo inmediatas que se dejan abiertas, tanto las principales que quedaban inicialmente fuera del alcance del presente proyecto como las mejoras secundarias que se ha decidido no incluir en esta ampliación de la plataforma.

9.1. RESULTADOS Y OBJETIVOS CUMPLIDOS

Este apartado enumera los principales hitos conseguidos durante la elaboración del presente proyecto, aprovechando para presentar al mismo tiempo las conclusiones extraídas de cada uno de ellos:

- Se ha demostrado la gran versatilidad de las librerías *Jini / Apache River*, siendo éstas fácilmente utilizables por cualquier aplicación *Java* para desplegar arquitecturas orientadas a servicios como la expuesta en capítulos anteriores. Se ha aprovechado la potencia de sus API's para hacer máximo uso de sus funcionalidades desde CAEAT, en el contexto de búsqueda y publicación de servicios, comunicación con los servidores de *Lookup*, despliegue de meta-servicios, etc.
- Se ha diseñado una semántica y una estructura de clases para los componentes aptos para ser publicados desde CAEAT. Dicha estructura es suficientemente canónica como para que un desarrollador de librerías pueda generar sus componentes con total desconocimiento de los detalles de implementación de la plataforma.
- Se ha diseñado un sistema de supervivencia, control y gestión de los servicios publicados suficientemente flexible como para hacer compatibles los servicios desplegados desde cualquiera de los canales a través de los cuales es posible hacerlo: la plataforma CAEAT, una máquina red-aceptante o un archivo autoejecutable. La obtención, utilización y gestión del servicio es transparente al método que se haya escogido para su despliegue.
- Se ha comprobado que el acceso concurrente a las variables de proceso alojadas en los servidores constituye un problema en un escenario como el propuesto en el presente proyecto, y se han tomado las medidas de protección adecuadas.
- Se ha dotado a la federación de la posibilidad de desplegar un servicio en cualquier punto de la misma que previamente se haya habilitado para tal fin. Este hito constituye el punto clave en la descentralización total de la gestión de los servicios, pues permite que cualquier usuario pueda lanzar servicios remotamente a cualquier punto de la red tal y como si se encontrase físicamente en dicho punto.
- Se ha comprobado que la edición concurrente de agregaciones de servicios ya publicadas representaba un problema, por lo que se ha habilitado un modo de edición concurrente de agregaciones que mantiene actualizadas las características de toda agregación de servicios mediante un sistema de envío y recepción de eventos de notificación remotos.
- Se ha experimentado exitosamente con la plataforma desarrollada en un entorno distribuido real, en un escenario conteniendo gran diversidad de sistemas operativos distintos. También se ha llevado a cabo un primer paso, puramente experimental, en la captura y tratamiento de magnitudes físicas del mundo real, demostrando de esta manera que la utilidad de la plataforma CAEAT como sistema SCADA es posible y encierra un gran potencial.

9.1.1. Objetivos secundarios

Se enumeran a continuación los hitos que no entraban en la planificación inicial del proyecto, pero que se han ido consiguiendo de manera paralela a los principales:

- La mejora continua de la interfaz gráfica de la plataforma CAEAT ha conseguido enriquecer enormemente la experiencia de usuario. Se ha buscado en todo momento hacer la herramienta lo más visual e intuitiva posible. Se han dado los primeros pasos hacia la personalización total de la interfaz de usuario, tanto en lo referente a estética como a herramientas de accesibilidad.
- Se ha implementado una primera versión del sistema de gestión de usuarios, necesario en un escenario de gestión de servicios como CAEAT / *SeNetComponents*. El sistema de gestión es fácilmente ampliable para poder definir diferentes juegos de permisos para cada usuario.
- Durante la elaboración del proyecto se han intentado aplicar nuevos patrones de diseño software con el fin de hacer el código fuente más reutilizable y de afianzar el dominio de las técnicas empleadas en el campo de la programación orientada a objetos. Asimismo, se han respetado y ampliado los patrones de software que ya habían sido aplicados en la fase previa a este proyecto.
- En la mejora continua de la interacción con el usuario, se ha modificado el aspecto de los componentes sobre el tapiz para que éstos reflejen de un vistazo ciertas informaciones útiles respecto de la agregación o servicio encapsulados. Se ha habilitado un sistema de asociación de imágenes a los componentes, consiguiendo hacer del ensamblaje de una agregación una experiencia más fluida, visual e intuitiva.

9.2. LÍNEAS DE TRABAJO ABIERTAS

Dado que se enmarca dentro de la elaboración de una ambiciosa plataforma software con múltiples vías de ampliación y mejora, este proyecto deja abiertas multitud de líneas de trabajo. Este apartado enumera las líneas futuras más inmediatas, dividiéndolas en dos grupos: por un lado, las vías principales que constituyen la ampliación estructural de la plataforma y los servicios y aplicaciones que la rodean; por el otro, las mejoras secundarias que aportarán valor añadido a la herramienta y que son implementables sobre la versión actual de la misma.

9.2.1. Ampliaciones principales

Se presentan a continuación las ampliaciones estructurales de la plataforma que dan inmediata continuidad al trabajo realizado durante el presente proyecto.

9.2.1.1. Almacenamiento mediante MySQL

Tal y como se detalló en el capítulo 2, el objetivo final de la plataforma CAEAT y los servicios que se pueden desplegar desde ella y para ella es el de servir como sistema de supervisión y adquisición de datos (SCADA). Este tipo de sistemas suelen poseer la capacidad de almacenar el histórico de eventos, datos y particularidades ocurridas en el transcurso del procesado que llevan a cabo, ordenando dichas eventualidades temporalmente (o atendiendo a otros criterios) y pudiendo consultar el registro a posteriori.

En el marco de CAEAT, resultará enormemente útil disponer de un sistema capaz de almacenar en una base de datos los datos adquiridos por una agregación, los distintos eventos lanzados por los componentes y, en general, cualquier dato útil generado durante la vida de una agregación de

componentes y susceptible de ser guardado en una base de datos. Para conseguir tal fin, se pretende integrar la plataforma *MySQL* con los componentes manejados por CAEAT.

MySQL es un sistema de gestión de bases de datos relacionales, *multithread* y multiusuario actualmente en propiedad de *Oracle Corporation* que se distribuye bajo licencia GNU GPL (para usos no comerciales) y bajo licencia privativa (para usos comerciales). Permite la ejecución y gestión de servicios que dan acceso a diferentes usuarios a un número indeterminado de bases de datos.

Figura 9.1: Logotipo del sistema de gestión de bases de datos *MySQL*

El objetivo de esta ampliación de la plataforma es el de poseer un componente de CAEAT (utilizable desde el tapiz y publicable como cualquier otro servicio) que se encargue de actuar de "puente" entre el usuario y una base de datos alojada en la máquina en la que se despliega el componente / servicio. Este componente permitirá escoger la base de datos con la cual se debe interactuar y admitirá como una de sus propiedades el dato a almacenar en ella (si se trata de una operación de escritura). Debe permitir realizar de manera sencilla las operaciones más habituales del lenguaje SQL (*select*, *update*, *insert*, etc.).

El componente a diseñar encapsulará los detalles de interacción con la base de datos y de implementación del lenguaje SQL. De esta manera se pretende liberar al usuario de la tarea de la construcción de sentencias SQL, pudiendo interaccionar con las bases de datos sin conocimiento alguno del lenguaje. Las diferentes operaciones a realizar y los parámetros de entrada y salida serán representados mediante propiedades externas (patillas) del componente a diseñar, gestionándose internamente la interacción con la base de datos, la resolución de errores, la adaptación del formato de los datos, etc.

Para más información acerca de la plataforma *MySQL*, consúltese su sitio oficial en la referencia [**19**].

9.2.1.2. Almacenamiento mediante Hibernate

Hibernate es un conjunto de librerías que complementan a la plataforma *Java* y que proporcionan un entorno de programación capaz de mapear un modelo de almacenamiento de datos basado en objetos (como el que manejan *Java* y todos los lenguajes de programación orientados a objetos) a un modelo de almacenamiento relacional (el utilizado actualmente por la gran mayoría de bases de datos y que habitualmente guarda la información organizándola en registros y campos).

La característica más potente ofrecida por *Hibernate* es la posibilidad de mapear directamente clases *Java* a las tablas de una base de datos, así como de convertir los tipos de datos manejados por *Java* a los equivalentes en el entorno SQL. También proporciona herramientas para facilitar la consulta y edición de los datos, generando las llamadas SQL necesarias y liberando al desarrollador de la gestión manual de la conversión de los datos y los objetos. Esto, al mismo tiempo, permite que la aplicación sea portable para trabajar con cualquier base de datos SQL.

Figura 9.2: Logotipo de la plataforma de almacenamiento *Hibernate*

Aplicadas a CAEAT, las librerías de *Hibernate* ayudarían a alcanzar un mayor nivel de abstracción en el sistema de almacenaje propuesto en el apartado anterior. *Hibernate* permitiría gestionar las bases de datos manejadas por los componentes creados en la ampliación anterior de manera mucho más eficiente y dotándolas de una mayor portabilidad. Esta ampliación, pues, debe ser vista como la evolución natural de la expuesta en el apartado anterior: los componentes capaces de interaccionar con bases de datos SQL acabarán utilizando *Hibernate* para tal fin.

Hibernate se distribuye mediante una licencia de código abierto GNU *Lesser General Public License*, y puede ser obtenido desde su sitio oficial (referencia [**20**]).

9.2.1.3. Implementación de protocolos estándar

Dado que el objetivo último de la plataforma CAEAT es el de actuar como sistema SCADA orientado a servicios, se hará necesaria la generación de componentes capaces de comprender e interactuar con los protocolos de comunicaciones más habituales que se usan en este campo de la industria. Estos componentes serán los encargados de realizar el tratamiento de los datos capturados por los dispositivos de campo, interpretándolos de una manera dependiente del protocolo y convirtiéndolos a un formato comprensible por la plataforma CAEAT (en la práctica, convirtiéndolos a tipos de datos del lenguaje de programación *Java*).

Algunos de los protocolos estándar más importantes en la actualidad en los entornos SCADA son los siguientes:

- ***Profibus***: el protocolo *Process Field Bus* define un estándar de comunicaciones para buses de campo. Fue desarrollado en 1989 por un consorcio de empresas e institutos y adoptado como norma europea en 1996. Es un protocolo propietario.
- ***Modbus***: desarrollado en 1979 para permitir la comunicación entre PLC's, y rápidamente convertido en un estándar *de facto*, este protocolo de comunicaciones goza de gran popularidad y aceptación en la actualidad debido a su sencillez y a ser de distribución pública. Existen versiones de *Modbus* para puerto serie y también para *Ethernet* (en este último caso se denomina *Modbus/TCP*).
- ***IEC 60870-5-104***: forma parte de un conjunto de normas que definen la monitorización y control de los sistemas automatizados de energía. En concreto, el comúnmente abreviado IEC 104 posee la gran ventaja de utilizar la red TCP/IP como entorno de trabajo, permitiendo (de la misma manera que *Modbus*) la transmisión sobre puerto serie o sobre *Ethernet*.

9.2.1.4. Interfaces móviles

Gracias a la proliferación de terminales móviles inteligentes con una capacidad de procesado cada vez superior, se desea experimentar con la adaptación y ejecución de la plataforma CAEAT y los servicios y accesorios que la rodean en entornos móviles.

El sistema operativo *Android*, de gran difusión en la actualidad y de código abierto, ofrece un entorno de programación muy favorable para el desarrollo y despliegue de aplicaciones elaboradas en el lenguaje de programación *Java*. La lógica de funcionamiento de la plataforma CAEAT puede ser adaptada a entornos móviles desde perspectivas diferentes y no mutuamente excluyentes:

- Las agregaciones autoejecutables generadas desde CAEAT podrían ser adaptadas para su exportación a entornos móviles. De esta manera, el dispositivo podría ejecutar y hacer uso de la agregación sin necesidad de disponer de una instalación de CAEAT.
- Toda la interfaz del editor CAEAT podría ser adaptada para su ejecución en un dispositivo móvil. La sencillez de uso de las pantallas táctiles encaja a la perfección con la metodología

de trabajo de CAEAT, basada en la inserción y conexión de elementos mediante operaciones de arrastre y dibujado de componentes gráficos.

- Sería posible diseñar una aplicación accesoria a CAEAT específica para que los dispositivos móviles actúen como clientes de las federaciones de servicios. Esta aplicación permitiría a un usuario unirse a una nueva federación y navegar entre los servicios publicados en ella, pudiendo obtener y visualizar los que considere oportunos, así como interactuar con las agregaciones mediante sus interfaces gráficas. Este escenario sería verdaderamente útil en un entorno SCADA inalámbrico en el cual los operarios interactuasen con los elementos mediante dispositivos portátiles.

9.2.1.5. Exportación como applet / servlet

Siguiendo la línea de la ampliación anterior, se desea buscar nuevas formas y entornos de ejecución para la plataforma CAEAT y los servicios y complementos que rodean a ésta. Una de estas nuevas formas de ejecución implicaría la interacción con un servidor mediante un *applet* / *servlet*. De esta manera, máquinas que no dispusiesen de una instalación de CAEAT podrían hacer uso de la plataforma a través de un navegador web (siempre que se dispusiese del permiso para ejercer dicho uso).

- ***Applet***: un *applet* es una programa que se puede incrustar en un documento HTML (es decir, en una página web) para su visualización y utilización desde cualquier navegador web. Cuando un cliente se conecta a una página web que contiene un *applet*, éste es descargado y el navegador inicia su ejecución. Por defecto, los *applets* están considerados como código no confiable y tienen un acceso restringido a los recursos del sistema, pero el cliente puede decidir conceder total permiso a la aplicación y ésta tendría a su alcance las mismas operaciones que cualquier otro programa. Se conseguiría así ofrecer acceso centralizado a CAEAT sin necesidad de la distribución física de sus paquetes a las máquinas cliente.
- ***Servlet***: la plataforma *Java* ofrece JSP (*JavaServer Pages*) como tecnología de generación de contenido dinámico en páginas web. Es el equivalente en la plataforma *Java* a tecnologías como PHP o ASP .NET. En combinación con la adaptación de CAEAT para su ejecución como applet, esta ampliación permitiría la ejecución de CAEAT como un entorno cliente-servidor. El servidor sería el encargado de llevar a cabo tareas de procesado como por ejemplo el mantenimiento de las sesiones de los clientes, la entrega a los clientes de los objetos correspondientes a las librerías que éstos pueden utilizar, etc.

9.2.1.6. Mejora en la gestión de usuarios

Al término del presente proyecto, se dispone de una primera versión del sistema de control de usuarios de la plataforma CAEAT. Como se ha explicado en capítulos anteriores, este control es vital puesto que todo servicio publicado es marcado con el nombre de su propietario, que dispone de permisos especiales de control y gestión sobre él.

Sin embargo, la gestión de usuarios se ha llevado a cabo de manera totalmente local, almacenando sus nombres y contraseñas en el fichero de configuración de la plataforma. Una primera mejora en el sistema de gestión de usuarios consistiría en crear un servicio de almacenaje de credenciales de usuario. Este servicio constituiría un nuevo meta-servicio de la plataforma CAEAT de manera similar a los servicios de reposición de librerías o las máquinas red-aceptantes.

Los servidores de usuarios, que deberían estar suficientemente redundados, almacenarían las credenciales de los usuarios en un fichero adecuadamente protegido. Desde la plataforma CAEAT sería posible acceder a estos servidores para dar de alta a nuevos usuarios o modificar los datos de los existentes. Al inicio de una sesión de trabajo, en la pantalla de introducción de *login* y *password*,

CAEAT buscaría en toda la federación de servicios un servidor de usuarios para poder verificar las credenciales que el usuario está introduciendo.

Pese a poder inicializar diferentes servidores de usuarios para reforzar la redundancia, los datos contenidos en todos ellos deben estar en sincronía. Además, se debe garantizar que dos usuarios no son dados de alta con el mismo nombre de *login*. Se debe permitir asimismo una utilización de CAEAT como usuario anónimo: no se tendrá propiedad sobre ninguna agregación y únicamente se podrá actuar como cliente, no como publicador de nuevos servicios.

La última ampliación de la plataforma en el campo de la gestión de usuarios sería la definición de diferentes perfiles de usuario, disponiendo cada uno de ellos de un juego distinto de permisos que definiría las acciones que éstos pueden llevar a cabo. Un usuario de perfil bajo, por ejemplo, podría tener vetada la publicación de servicios, pero sí su utilización. Un usuario de perfil alto podría tener habilitada la gestión de servicios ajenos a él. Incluso podría existir la figura del superusuario, capaz de llevar a cabo cualquier acción y de gestionar los permisos del resto de usuarios. La información de los juegos de permisos asociados a cada usuario se almacenaría en los ficheros de los servidores de usuarios junto con sus credenciales.

9.2.2. Mejoras secundarias

Se presentan a continuación las mejoras secundarias que no constituyen ampliaciones estructurales importantes de la plataforma y los servicios que la rodean, pero que la dotarán de valor añadido, facilidad de uso y versatilidad.

9.2.2.1. Copy-paste de objetos remotos

En la versión actual de la plataforma no se ha permitido la realización de operaciones de copiado, cortado y pegado sobre los objetos remotos del tapiz (ya sean éstos servicios simples o agregaciones complejas de componentes). En el caso de los componentes locales, la copia de elementos se limitaba al duplicado de los objetos implicados y en la inserción de una segunda referencia en el tapiz. En el caso de los componentes remotos, este proceso no es aplicable puesto que se deben realizar operaciones intermedias: obtención de los *proxies* de los servicios implicados, registro de la plataforma CAEAT como clienta de los distintos servicios, actualización inicial de los valores de las propiedades, etc.

Estas operaciones pueden llevarse a cabo automáticamente mediante código, permitiendo al usuario continuar utilizando las funciones de copiado de manera independiente a la naturaleza de los objetos que se están manejando (locales o remotos). Ante el copiado de una agregación que contiene ambos tipos de objeto, se deberá realizar un tratamiento claramente diferenciado para los locales y para los remotos.

Pese a que el cortado y pegado de componentes remotos no presenta esta problemática (ya que las instancias de los objetos implicados son las mismas y únicamente cambia su lugar de inserción), se ha optado por prohibirla también por coherencia con la prohibición de las operaciones de copiado.

9.2.2.2. Modo debug / paso a paso

Durante la construcción de agregaciones de componentes complejas con una gran cantidad de niveles de anidamiento se pueden producir errores de origen humano no detectados que lleven a un mal funcionamiento de la agregación cuando ésta se encuentre finalizada y operativa. La naturaleza de estos errores puede ser muy sutil: por ejemplo, la creación incorrecta de una conexión puede provocar que un componente no reciba unos eventos vitales para su funcionamiento, resultando en

consecuencia en una agregación defectuosa que no lleva a cabo el procesado para la cual fue diseñada.

Si como se ha comentado la agregación afectada cuenta con una gran cantidad de componentes y niveles de encapsulación, la localización y corrección de errores tan sutiles como éste puede resultar tediosa y requerir de una gran cantidad de tiempo. En este contexto resultaría de gran utilidad la existencia de un modo de ejecución de *debug* para la plataforma CAEAT. Puesto que las agregaciones de componentes generadas mediante CAEAT realizan su procesado gracias a las cadenas de eventos iniciadas en un componente y transmitidas a lo largo del resto de componentes conectados a éste, la finalidad principal del modo de *debug* sería la de "romper" esta cadena de eventos para permitir al usuario desglosar paso a paso los cambios en los valores de las propiedades de los distintos componentes.

Para conseguir tal fin, se deberá permitir al usuario la inserción de marcas de interrupción o *breakpoints* en los componentes. Ante la detección del cambio de valor de una de las propiedades, y como paso previo al lanzamiento del evento que dará continuidad a la cadena, la ejecución se detendrá momentáneamente y se informará al usuario del cambio acontecido. Una manera eficaz de realizar esta operación consistiría en la apertura de un cuadro de diálogo que mostrase la información relevante acerca del evento ocurrido. La ejecución normal de la agregación se reanudaría en el momento en que el usuario cerrase dicho diálogo.

Mediante la inserción recursiva de *breakpoints* en los componentes sospechosos de ser los generadores del error y la ejecución paso a paso, el tiempo empleado en la detección de errores de difícil localización se vería reducido enormemente.

9.2.2.3. Mejoras en la accesibilidad

Este proyecto ha realizado un primer paso para dotar a la plataforma CAEAT de mecanismos de accesibilidad para las personas con deficiencias visuales. En concreto, tal y como se detalla en el anexo D.2, se han implementado interfaces de usuario de alto contraste, que hacen uso de juegos de colores tales que sean fácilmente diferenciables por personas con dificultades para la percepción de la gama cromática.

Los siguientes pasos a ejecutar para conseguir una total aplicación de los mecanismos tradicionales de accesibilidad software implican una revisión y replanteo más profundos de la interfaz gráfica de la herramienta. Una de las medidas aplicadas consistiría en la selección de tamaños de fuente más grandes para los diferentes textos mostrados en los menús, botones, etc. Otra mejora en este campo consistiría en el aumento del tamaño de los elementos con los cuales el usuario ha de interactuar: *checkboxes*, campos de texto, patillas y conexiones de los componentes, etc.

En algunos casos estas medidas obligarían al rediseño del *layout* usado en la interfaz gráfica de CAEAT, que volvería al estado normal en el momento en que el usuario lo desease. Todas las mejoras introducidas en el campo de la accesibilidad deberán poder ser activadas en tiempo de ejecución desde el cuadro de selección de opciones implementado (véase anexo D.4), y del mismo modo la interfaz normal de trabajo deberá ser restaurada en el momento en que el usuario así lo indique mediante el mencionado cuadro de selección.

9.2.2.4. Zoom y tamaño del tapiz

El presente proyecto ha respetado el criterio de diseño usado en la versión inicial de la plataforma CAEAT, que establece el tamaño del tapiz de trabajo en 800x1000 píxeles. Los componentes que en él se alojan tienen un tamaño de 80x50 píxeles más un margen de guarda invisible para evitar el solapamiento de sus patillas y sus iconos descriptivos. En un entorno de trabajo real, en el cual se

crearán agregaciones densamente pobladas de componentes, el tapiz de CAEAT podría llegar a quedarse pequeño.

Sin embargo, un simple aumento en el tamaño del tapiz lo haría inmanejable, puesto que el fragmento de la zona de trabajo visible desde la interfaz de CAEAT únicamente representa una pequeña parte de él. Un tapiz grande obligaría a hacer un uso excesivo de las barras de desplazamiento y dificultaría la localización de componentes alejados de la zona de visualización actual. El aumento del tamaño del tapiz debe ir acompañado de la opción de abrir el campo de visión sobre la zona de trabajo para poder visualizarla completamente a una escala más reducida.

En este nuevo modo de visualización los componentes se representarían a una escala mucho más reducida. Una solución altamente intuitiva para el usuario consistiría en representarlos mediante su icono descriptivo de tamaño 16x16 píxeles. No sería posible representar las patillas de los componentes, pero sí se representarían esquemáticamente las conexiones entre ellos. Se dibujaría un rectángulo alrededor del cursor para indicar la zona que se ampliará al realizar nuevamente *clic* sobre la vista a escala reducida del tapiz.

El modo de visualización a escala reducida no sería apropiado para trabajar con la agregación, simplemente se ofrecería al usuario para que pudiese restaurar el nivel de *zoom* normal en la zona del tapiz en la cual desee continuar su trabajo.

9.2.2.5. Traducción de la plataforma

Una mejora a realizar sobre la plataforma para poder lanzarla en un entorno de producción consiste en habilitar la selección del idioma en el cual se muestran los textos de pantalla. Hasta el momento, el proyecto CAEAT ha escogido el castellano como lengua de interacción con el usuario. Todos los textos que se muestran por pantalla en la interfaz de la plataforma (menús, botones, mensajes de información, etc.) se han insertado de manera literal en el código fuente de la aplicación.

Para permitir la selección de idiomas alternativos en tiempo de ejecución desde el cuadro de selección de opciones implementado (ver anexo D.4), la información textual a mostrar por pantalla debe poder ser cargada de manera dinámica desde una fuente externa. Generalmente, la localización e internacionalización del software debe poder realizarse rápidamente sin necesidad de recompilar el código fuente ni de utilizar un ejecutable distinto.

Los textos y mensajes mostrados por la plataforma CAEAT se almacenarían en un fichero textual específicamente creado para tal fin, que se alojaría en el sistema de ficheros de la propia plataforma. Los textos estarían organizados siguiendo un formato de fichero de configuración, mediante un mapa clave - valor en el cual sería posible buscar de manera dinámica cualquier texto deseado a partir de la clave apropiada. Por ejemplo, la clave “alarm.spanish” retornaría un mensaje de alarma en español, mientras que “alarm.english” retornaría su equivalente en inglés.

Una variable global de proceso almacenaría el lenguaje vigente en cada momento para poder generar claves como las mostradas rápidamente. Nótese que de esta manera, la inclusión de un nuevo lenguaje para la plataforma no implica la más mínima modificación en el código fuente de la misma: únicamente se debería ampliar el fichero de mensajes para traducir al nuevo idioma la colección de asociaciones clave – valor contenidas en dicho fichero.

9.2.2.6. Mejoras en el asistente de creación de interfaces gráficas

Actualmente, el asistente de creación de interfaces gráficas para las agregaciones consiste únicamente en una rejilla sobre la cual los componentes gráficos son representados simbólicamente mediante rectángulos amarillos. La creación de la interfaz gráfica de la agregación se realiza mediante el posicionamiento y redimensionado de dichos rectángulos sobre la rejilla. El asistente

puede ser mejorado enormemente para facilitar su utilización al usuario mediante las tres acciones siguientes:

- Utilización de la imagen asociada al componente gráfico en lugar de una representación esquemática mediante un rectángulo amarillo. Esta mejora permitiría a los usuarios hacerse una idea más acertada del aspecto final de la interfaz al mismo tiempo que la están creando, haciendo innecesaria la operación de cerrar el asistente y usar la función de previsualización de la interfaz creada.
- Posibilidad de edición del contenido de los paneles desde la ventana del asistente. En la actualidad, los elementos gráficos contenidos en un panel únicamente son modificables desde el tapiz, a través del *Customizer* del componente Panel, lo que obliga a cerrar el asistente gráfico en caso de querer añadir o retirar elementos del/los panel/es bajo edición.
- Posicionamiento automático de los elementos sobre el *layout* del panel principal. En la actualidad, los elementos gráficos se insertan en el punto (0,0) de su panel contenedor. Esto provoca que todos se muestren en un primer momento en la esquina superior izquierda de la interfaz, y es responsabilidad del usuario la correcta colocación de los mismos. Un algoritmo de ubicación automática de los elementos en el momento de su inserción mejoraría enormemente la experiencia de usuario, puesto que éste recibiría una sugerencia de presentación de los elementos gráficos y únicamente debería reubicar aquéllos que considerase oportunos.

9.2.2.7. Generación de librerías a partir de esquemas

Actualmente, el proceso de creación de una librería de componentes apta para su utilización con CAEAT implica la codificación manual de las diferentes clases *Java* implicadas en el proceso, cuya estructura general y funciones fueron detalladas en los capítulo 2 y 3 y cuyo código es detallado en profundidad en los anexos A y B. Además, por el momento, una librería de componentes contiene *beans* sencillos que son utilizables de manera individual por la plataforma. Resultaría interesante la posibilidad de ofrecer en una librería componentes formados por multitud de *beans* encapsulados en una "caja negra"; en otras palabras, ofrecer una agregación generada mediante CAEAT como si fuese cualquier otro componente integrante de la librería.

Tras la implementación de esta mejora, y en el momento de la finalización del ensamblaje de una nueva agregación en CAEAT, se podrá optar por cuatro opciones no mutuamente excluyentes:

- Guardado de la agregación en el sistema de archivos mediante un fichero *aec*.
- Exportación a un fichero autoejecutable *jar*.
- Inicialización como servicio y publicación.
- Inclusión en una librería de componentes generada por el usuario.

En caso de optar por la nueva opción propuesta por la presente mejora, se generaría un archivo *jar* al cual se copiarían todos los recursos relacionados con los *beans* simples que forman parte de la agregación. Se daría al usuario la posibilidad de elegir las propiedades de la agregación visibles desde el exterior del componente (las patillas), de la misma manera en que se realizaba a la hora de publicarla. Se guardaría asimismo la estructura interna de la agregación en un archivo *aec* con el formato expuesto en el capítulo 5, y algunas informaciones que ayudarán a identificar al componente (nombre, icono, etc.). Por último, se generaría automáticamente un archivo *Manifest* con la descripción de los componentes contenidos en la librería (la creación de este archivo es manual si la creación de la librería también lo es).

Además de la creación de la librería, se deberá dar la posibilidad de añadir la agregación generada a una librería ya existente. En este caso, el archivo *jar* de la librería sería editado para incluir en él

todos los elementos descritos anteriormente. Para evitar conflictos, el usuario únicamente podrá añadir componentes nuevos a archivos *jar* generados por él mismo: las librerías generadas por terceros y obtenidas desde la red deberán permanecer inmutables.

Los componentes así creados serán mostrados en el tapiz de CAEAT de la misma manera que si fuesen *beans* simples, pero en realidad encapsularían agregaciones complejas. Dado que el usuario no podrá navegar por su contenido, se deberá implementar un *Customizer* genérico que permita editar visualmente las propiedades del componente tal y como si se tratase de un bean. El *Customizer* se generará de manera dinámica atendiendo a los tipos de datos de las propiedades externas (cuadros de texto para los *Strings*, *checkboxes* para los *boolean*, etc.).

Una librería generada mediante esta técnica debe poder ser utilizable de manera idéntica a las librerías tradicionales. Al insertarla en tiempo de ejecución a CAEAT, éste extraerá todos los recursos y mostrará al usuario los componentes como si de *beans* simples se tratase. Se ocultará en todo momento el hecho de que algunos componentes pueden constituir *beans* simples (como en las librerías utilizadas durante el presente proyecto) o agregaciones complejas (las resultantes tras esta ampliación).

9.2.2.8. Enriquecimiento de los objetos Proxy

Tal y como se detallará en el anexo B, el entorno *Jini / Apache River* ofrece multitud de herramientas para dotar al proceso de descarga dinámica de objetos y realización de llamadas remotas a través de ellos de múltiples características adicionales. En concreto, la generación de *proxies* hacia los servicios desplegados puede incluir restricciones que obliguen a que las llamadas remotas se realicen bajo unas condiciones estrictas: integridad, autentificación por parte del cliente o del servidor, confidencialidad mediante el cifrado de los mensajes intercambiados, etc.

El presente proyecto ha generado los *proxies* de los servicios de manera que las mencionadas restricciones sean fácilmente aplicables, pero no ha experimentado con la utilización de ninguna de ellas. Resultaría interesante la realización de distintas pruebas con estas herramientas, que generarían tipos de *proxies* distintos, para discernir cuáles serán útiles en el futuro entorno de ejecución real de la plataforma CAEAT.

A. IMPLEMENTACIÓN COMPLETA DE UN BEAN

En este anexo se presenta y comenta a modo de ejemplo el código fuente completo de uno de los *beans* programados para CAEAT. El bean escogido pertenece a la Librería Matemática, y tiene por función la suma de dos números enteros. Recuérdese que la estructura necesaria para el diseño e implementación de un *bean* simple apto para su uso con la plataforma CAEAT (pero no para su publicación como servicio) envuelve tres clases: la del propio *bean*, su `BeanInfo` y su `Customizer`.

A.1. BEAN

```
package sener.matematica.aritmetica;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

public class Sumador implements Serializable {

	private static final long serialVersionUID = 7524905699921247210L;
	private int a;
	private int b;
	private int resultado;
	private boolean suma;
	private PropertyChangeSupport cambios;

	public Sumador() {
		a = 0;
		b = 0;
		resultado = 0;
		suma = false;
		cambios = new PropertyChangeSupport(this);
	}

	public int getA() {
		return a;
	}

	public void setA(int a) {
		int aAnt = this.a;
		if (aAnt != a) {
			this.a = a;
			cambios.firePropertyChange("a", aAnt, a);
		}
	}

	public int getB() {
		return b;
	}

	public void setB(int b) {
		int bAnt = this.b;
		if (bAnt != b) {
			this.b = b;
			cambios.firePropertyChange("b", bAnt, b);
		}
	}

	public int getResultado() {
		return resultado;
	}

	public boolean isSuma() {
		return suma;
	}
```

```
        public void realizaSuma(boolean suma) {
                if (suma) {
                        resultado = a + b;
                        cambios.firePropertyChange("resultado", null, resultado);
                        cambios.firePropertyChange("suma", false, suma);
                        cambios.firePropertyChange("suma", true, false);
                }
                this.suma = false;
        }

        public void addPropertyChangeListener(PropertyChangeListener l) {
                cambios.addPropertyChangeListener(l);
        }

        public void removePropertyChangeListener(PropertyChangeListener l) {
                cambios.removePropertyChangeListener(l);
        }

        public PropertyChangeListener[] getPropertyChangeListeners() {
                return cambios.getPropertyChangeListeners();
        }

        public PropertyChangeListener[] getPropertyChangeListeners(String
                propertyName) {
                return cambios.getPropertyChangeListeners(propertyName);
        }
}
```

Código A.1: Implementación del *bean* Sumador

La clase que implementa el *bean* declara las cuatro propiedades que serán visibles desde el exterior en forma de patillas: `a`, `b`, `suma`, `resultado`. El atributo `serialVersionUID` sirve para mantener un control de versiones de los objetos serializados, mientras que el objeto `cambios` se utiliza para lanzar los eventos apropiados cuando una propiedad es modificada.

Se puede observar cómo la clase proporciona métodos *get* y *set*, genéricamente llamados *getters* y *setters*, para el manejo de las propiedades. Es posible declarar métodos que no sigan la semántica establecida por *JavaBeans*, como es el caso de *realizaSuma*, siempre que se especifique adecuadamente en la clase `BeanInfo` (ver siguiente apartado). Por último, cabe destacar que el *bean* también ofrece métodos para añadir y eliminar *listeners* interesados en los cambios de valor de las propiedades. Estos métodos son usados por CAEAT para añadir como *listeners* a todos los componentes "conectados" a éste.

A.2. BEANINFO

```
package sener.matematica.aritmetica;

import java.awt.Image;
import java.beans.BeanDescriptor;
import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;

public class SumadorBeanInfo extends SimpleBeanInfo {

        private static Class<?> beanClass = Sumador.class;
```

```
	public PropertyDescriptor[] getPropertyDescriptors() {
		PropertyDescriptor[] pd = null;
		try {
			PropertyDescriptor a = new PropertyDescriptor("a", beanClass);
			a.setBound(true);
			PropertyDescriptor b = new PropertyDescriptor("b", beanClass);
			b.setBound(true);
			PropertyDescriptor resultado = new PropertyDescriptor("resultado",
			beanClass, "getResultado", null);
			resultado.setBound(true);
			PropertyDescriptor suma = new PropertyDescriptor("suma",
			beanClass, "isSuma", "realizaSuma");
			suma.setBound(true);
			pd = new PropertyDescriptor[] { a, b, resultado, suma };
		} catch (IntrospectionException e) {
			e.printStackTrace();
		}
		return pd;
	}
```

Código A.2: Implementación de la clase *BeanInfo* del bean Sumador

El método *getPropertyDescriptors* se usa para definir las propiedades del *bean*. CAEAT utiliza la información proporcionada por este método para determinar los atributos, y por lo tanto las patillas, que contiene un componente. Cabe destacar que es opcional especificar el nombre de los métodos *getter* y *setter* de cada atributo, pero que se debe ser consecuente con el nombre escogido en la clase del *bean*. Si no se especifica nombre, la API dará por hecho que estos métodos siguen la semántica definida por el estándar (consúltese referencia [**5**]). Es posible declarar un atributo como sólo de lectura ó solo de escritura si su método *set* / *get* (respectivamente) se especifica en el `BeanInfo` como *null*.

Por último, se debe destacar la sobreescritura del método *getIcon*, extremadamente útil en una plataforma de edición gráfica como CAEAT. Este método se usa para obtener el icono que se muestra en la paleta y en otros lugares de la herramienta. El icono habrá sido proporcionado en el archivo *jar* de la librería por el creador de la misma.

A.3. CUSTOMIZER

```
package sener.matematica.aritmetica;

import java.awt.Dimension;
import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.beans.Customizer;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class SumadorCustomizer extends JPanel implements Customizer, MouseListener,
	FocusListener {

	private static final long serialVersionUID = -6136771209016660993L;
	private Sumador target;
	private JTextField aTF;
	private JTextField bTF;
	private JTextField resultadoTF;
	private JButton sumar;
```

```
	public SumadorCustomizer() {
		setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
		setAlignmentX(LEFT_ALIGNMENT);
		JPanel abPanel = new JPanel();
		abPanel.setLayout(new BoxLayout(abPanel, BoxLayout.X_AXIS));
		abPanel.setAlignmentX(LEFT_ALIGNMENT);
		abPanel.add(new JLabel("a"));
		addSpacer(abPanel);
		aTF = new JTextField();
		aTF.addFocusListener(this);
		abPanel.add(aTF);
		addSpacer(abPanel);
		abPanel.add(new JLabel("b"));
		addSpacer(abPanel);
		bTF = new JTextField();
		bTF.addFocusListener(this);
		abPanel.add(bTF);

		add(abPanel);
		addSpacer(this);

		abPanel = new JPanel();
		abPanel.setLayout(new BoxLayout(abPanel, BoxLayout.X_AXIS));
		abPanel.setAlignmentX(LEFT_ALIGNMENT);
		sumar = new JButton("Sumar");
		sumar.addMouseListener(this);
		sumar.setAlignmentX(LEFT_ALIGNMENT);
		abPanel.add(sumar);
		addSpacer(abPanel);
		addSpacer(abPanel);
		abPanel.add(new JLabel("Resultado"));
		addSpacer(abPanel);
		resultadoTF = new JTextField();
		resultadoTF.setEnabled(false);
		abPanel.add(resultadoTF);

		add(abPanel);
		addSpacer(this);
	}

	public void setObject(Object obj) {
		target = (Sumador) obj;
		aTF.setText("" + target.getA());
		bTF.setText("" + target.getB());
		resultadoTF.setText("" + target.getResultado());
		revalidate();
	}

	private void addSpacer(JPanel panel) {
		panel.add(Box.createRigidArea(new Dimension(12, 12)));
	}

	public void mouseClicked(MouseEvent e) {
		if (e.getComponent().equals(sumar)) {
			target.realizaSuma(true);
		}
		setObject(target);
	}

	public void mouseEntered(MouseEvent e) {
	}

	public void mouseExited(MouseEvent e) {
	}

	public void mousePressed(MouseEvent e) {
	}

	public void mouseReleased(MouseEvent e) {
	}

	public void focusGained(FocusEvent e) {
	}
```

```
        public void focusLost(FocusEvent e) {
                JTextField tf = (JTextField) e.getSource();
                String text = tf.getText();
                try {
                        int valor = Integer.parseInt(text);
                        if (tf == aTF) {
                                target.setA(valor);
                        } else {
                                target.setB(valor);
                        }
                } catch (NumberFormatException nfe) {
                }
        }
}
```

Código A.3: Implementación de la clase *Customizer* del bean Sumador

Como se puede observar, la mayor parte del código de un `Customizer` corresponde al diseño de su interfaz gráfica. La clase anterior permite visualizar el diálogo del `Customizer` mostrado en el capítulo 2 (figura 2.10) al realizar doble *clic* sobre el componente Sumador en el tapiz de CAEAT. Cabe destacar la captura de las acciones de *mouse* para actualizar adecuadamente tanto la interfaz gráfica del diálogo como las propiedades del componente, si éstas han sido cambiadas.

Las especificaciones únicamente requieren la implementación del método *setObject*, para poder inicializar adecuadamente la interfaz gráfica y asociar al `Customizer` el objeto que representa al *bean*. Al margen de esto, las especificaciones dan total libertad en cuanto al diseño de la interfaz gráfica del `Customizer` y sus mecanismos de actualización, un hecho muy positivo en una herramienta de manejo visual como CAEAT.

B. IMPLEMENTACIÓN COMPLETA DE UN SERVICIO REMOTO

En el capítulo 3 se presentó la nueva estructura de clases *Java* que un servicio debe proporcionar para poder ser compatible con su exportación, publicación, obtención y uso por parte de la herramienta de edición CAEAT. Este anexo profundiza en el diseño e implementación de dichas clases, de la misma manera que en el anexo A se presentaron los detalles de implementación de un *bean* compatible con CAEAT para su utilización únicamente en el ámbito local.

El conjunto inseparable de las tres clases implicadas en el diseño de un *bean* comprensible por la plataforma CAEAT estaba formado por la clase del propio *bean*, un `BeanDescriptor` y un `Customizer`. Para permitir la publicación del *bean* como servicio *Jini / Apache River*, se ha diseñado un nuevo conjunto de clases formado por cinco nuevos elementos: la interfaz remota, el *proxy*, el *wrapper* (u objeto remoto), el servidor y el publicador.

No es obligatorio que todo *bean* implemente las ocho clases mencionadas. Una librería de componentes compatibles con CAEAT puede contener al mismo tiempo:

- *Beans* que únicamente implementen las tres clases iniciales, y que por lo tanto sólo sean compatibles para su uso en el entorno local de CAEAT.
- *Beans* que implementen las ocho clases, y que por lo tanto sean compatibles tanto para su uso en modo local como para su exportación y publicación como servicio *Jini / Apache River*.

La plataforma CAEAT conoce y distingue en todo momento el conjunto de *beans* compatibles con la publicación. Una agregación contenida en el tapiz es apta para publicación sólo si todos sus componentes lo son (o son ya remotos). De esta manera es posible informar al usuario de que una agregación no se puede publicar en caso de que ésta contenga algún *bean* no compatible con las operaciones remotas. También es posible, en el momento de la exportación a un autoejecutable *jar*, decidir si se le debe pedir al usuario la definición de la tabla de propiedades externas de la agregación (paso que únicamente tiene sentido llevar a cabo si la agregación es publicable).

El código fuente mostrado en el presente apéndice corresponde al de un servicio de pruebas sencillo llamado “Contador de Letras”, que calcula y devuelve el número de letras contenido en un mensaje de tipo *String*.

B.1. INTERFAZ REMOTA

La interfaz descriptora del servicio define sus métodos *getter* y *setter* que serán accesibles por los clientes, además de otros métodos propios de los servicios diseñados para CAEAT, que sirven para su gestión y mantenimiento. La interfaz de un servicio remoto extiende la interfaz de RMI `Remote` como requisito imprescindible, puesto que todo objeto susceptible de ser exportado para la recepción de llamadas remotas debe implementar `Remote`.

De entre las nuevas clases que se describen en este anexo, la implementación de la interfaz del servicio es obligatoria para las dos siguientes:

- ***Proxy***: los objetos que sean instancia de esta clase actúan de “puente” entre los clientes y la entidad que proporciona el servicio real, por lo que llevan a cabo las llamadas remotas al servidor. Los métodos ofrecidos por el *Proxy* deben ser, por ese motivo, los mismos que ofrece el servicio (tanto los *getter* y *setter* como los de gestión del servicio), por lo que se obliga a esta clase a la implementación de la interfaz remota.
- **Servidor**: esta clase aloja el procesado y las propiedades reales del servicio, por lo cual se debe garantizar que proporciona una implementación real a todos los métodos que se quieren poder a disposición de los clientes. Además, como clase que se exporta para la recepción de llamadas remotas, debe implementar la interfaz `Remote`, por lo que la

implementación de la interfaz del servicio, cuyo código completo se muestra a continuación, es doblemente obligada.

```
package sener.publico.interfaces;

public interface ContadorLetrasInterface extends Remote {

	//Métodos propios del servicio ContadorLetras:
	public String getMensaje() throws RemoteException;
	public void setMensaje(String mensaje) throws RemoteException;
	public boolean isContajeAutomatico() throws RemoteException;
	public void setContajeAutomatico(boolean contajeAutomatico) throws RemoteException;
	public boolean isCuenta() throws RemoteException;
	public void realizaCuenta(boolean contar) throws RemoteException;
	public int getLetras() throws RemoteException;

	//Métodos de gestión y mantenimiento del servicio:
	public boolean isAlive() throws RemoteException;
	public void setAlive(boolean alive) throws RemoteException;
	public long register(RemoteEventListener l) throws RemoteException;
	public void unregister(String nombreInstancia, RemoteEventListener rel)
		throws RemoteException;
	public void finish() throws RemoteException;
	public void setInitialProperties(Object[] props) throws RemoteException;
	public void setCSMListener(ActionListener CSM) throws RemoteException;
	public void stopRemotely(ServiceID id) throws RemoteException;
	public long getExpiration() throws RemoteException;
	public void setExpiration(ServiceID id, long exp) throws RemoteException;
	public boolean resetService(ServiceID id, long duration, RemoteEventListener owner)
		throws RemoteException;

}
```

Código B.1: Implementación completa de la clase *ContadorLetrasInterface.class*

Nótese que *ContadorLetras* tiene cuatro propiedades básicas: `mensaje`, `contajeAutomatico`, `cuenta` y `letras`. Se ha decidido que todas ellas sean tanto de lectura como de escritura excepto `letras`, que únicamente es de lectura ya que no proporciona un método *setter*. En el entorno CAEAT esto se traducirá en una patilla únicamente de salida en el componente en cuestión. Los métodos de gestión y mantenimiento del servicio serán detallados en el apartado dedicado al Servidor.

B.2. PROXY

El *proxy* es la clase encargada de realizar las llamadas remotas al servidor y devolver los resultados a la entidad que ha realizado la llamada. Junto con el *wrapper*, es una de las dos clases cuyas instancias se registran en los servidores de *Lookup* para ofrecer a los clientes las herramientas necesarias de interacción con el servicio.

En el momento de la exportación de un objeto *Java* para la recepción de llamadas remotas (operación que se lleva a cabo con los objetos que son instancia de la clase *Servidor*), se genera un objeto *proxy* que implementa la interfaz del servicio y que puede ser utilizado para acceder a dicho servicio. Pese a que este objeto ya podría ser usado como *proxy* a registrar en los servidores LUS, el estándar *Jini* recomienda encarecidamente encapsularlo en un objeto *proxy* customizado. Las razones que llevan a esta práctica son principalmente dos:

- Evitar que el objeto *Proxy* sea un mero puente entre clientes y servidores y dotarlo de las capacidades de un *smart proxy*; es decir, permitir que el objeto *proxy* sea capaz de ejecutar código internamente antes y después de realizar la llamada remota.
- Implementar interfaces del entorno *Jini / Apache River* que permitan al *proxy* ofrecer funcionalidades adicionales como autenticación, integridad y confidencialidad de las comunicaciones.

El código completo del *proxy* del servicio Contador de Letras se presenta a continuación.

```
package sener.publico.proxies;

public class ContadorLetrasProxy extends ServiceInfo implements ContadorLetrasInterface,
        Serializable, RemoteMethodControl {

        private static final long serialVersionUID = 1002899674354125874L;
        final ContadorLetrasInterface inter;

        public ContadorLetrasProxy(ContadorLetrasInterface inter) {
                this.inter = inter;
                this.name = "Contador de letras en un mensaje";
                this.version = "1.0";
                this.manufacturer = "Sener";
                this.vendor = "Sener";
        }

        public static ContadorLetrasProxy crear(ContadorLetrasInterface inter) {
                return new ContadorLetrasProxy(inter);
        }

        public String getMensaje() throws RemoteException {
                return inter.getMensaje();
        }

        public void setMensaje(String mensaje) throws RemoteException {
                inter.setMensaje(mensaje);
        }

        public boolean isContajeAutomatico() throws RemoteException {
                return inter.isContajeAutomatico();
        }

        public void setContajeAutomatico(boolean contajeAutomatico)
                throws RemoteException {
                inter.setContajeAutomatico(contajeAutomatico);
        }

        public boolean isCuenta() throws RemoteException {
                return inter.isCuenta();
        }

        public void realizaCuenta(boolean contar) throws RemoteException {
                inter.realizaCuenta(contar);
        }

        public int getLetras() throws RemoteException {
                return inter.getLetras();
        }

        public boolean isAlive() throws RemoteException {
                return inter.isAlive();
        }

        public void setAlive(boolean alive) throws RemoteException {
                inter.setAlive(alive);
        }

        public long getExpiration() throws RemoteException {
                return inter.getExpiration();
        }

        public void setExpiration(ServiceID id, long exp) throws RemoteException {
                inter.setExpiration(id,exp);
        }

        public boolean resetService(ServiceID id, long duration, RemoteEventListener owner)
                throws RemoteException {
                return inter.resetService(id, duration, owner);
        }
```

```
        public long register(RemoteEventListener l) throws RemoteException {
                return inter.register(l);
        }

        public void unregister(String nombreInstancia, RemoteEventListener rel)
                throws RemoteException {
                inter.unregister(nombreInstancia, rel);
        }

        public void finish() throws RemoteException {
                inter.finish();
        }

        public void setCSMListener(ActionListener CSM) {
                //Método no usado por los proxies, únicamente por los servidores
        }

        public void stopRemotely(ServiceID id) throws RemoteException {
                inter.stopRemotely(id);
        }

        public void setInitialProperties(Object[] props) throws RemoteException {
                inter.setInitialProperties(props);
        }

        public MethodConstraints getConstraints() {
                return ((RemoteMethodControl) inter).getConstraints();
        }

        public RemoteMethodControl setConstraints(MethodConstraints mc) {
                return new ContadorLetrasProxy(
                        (ContadorLetrasInterface) ((RemoteMethodControl)
                                inter).setConstraints(mc));
    }
}
```

Código B.2: Implementación completa de la clase *ContadorLetrasProxy.class*

El *proxy* debe implementar la interfaz `Serializable` dado que se trata de un objeto que será serializado y transmitido por la red. Ello obliga a establecer un atributo `serialVersionUID` para llevar control de las distintas versiones de la clase de la cual el objeto *proxy* es una instancia. Por las razones anteriormente expuestas, el *Proxy* también implementa la interfaz del servicio. Como funcionalidad añadida, los *proxies* de las librerías diseñadas durante el presente proyecto heredan de la clase `ServiceInfo`, que contiene una serie de campos públicos (*name*, *manufacturer*, *version*, etc.) que son de ayuda a la hora de mostrar información acerca del servicio en formato agradable en el momento de las búsquedas.

Las funcionalidades adicionales ofrecidas por los *proxies* de *Jini / Apache River* se consiguen, en parte, mediante la implementación de la interfaz `RemoteMethodControl`. Mediante los dos métodos implementados por dicha interfaz es posible establecer juegos de restricciones sobre las llamadas remotas realizadas por el objeto *proxy*. Las restricciones son representadas mediante clases del lenguaje de programación *Java* pertenecientes a las librerías de *Jini / Apache River*. Algunas de ellas son:

- `ClientAuthentication`, `ServerAuthentication`: obliga a la autentificación por parte del cliente o del servidor, respectivamente.
- `Confidentiality`: obliga a la realización de las comunicaciones mediante un mecanismo que garantice su confidencialidad (típicamente, haciendo uso de encriptación).
- `Integrity`: obliga a la comprobación de la integridad de los mensajes recibidos mediante las llamadas remotas.

Este proyecto no ha experimentado con el establecimiento de restricciones sobre las llamadas remotas realizadas en los *proxies* de los servicios. Sin embargo, todos los *proxies* generados durante su elaboración admiten el establecimiento de restricciones especiales como las expuestas, dado que en un entorno de producción será conveniente obligar a la utilización de algunas de ellas. La lista completa de restricciones y las especificaciones completas de los juegos de clases implicadas en este proceso se pueden consultar en la documentación de referencia de Apache River (referencia [**3**]).

Una última funcionalidad que se podría ofrecer sería la capacidad, por parte del cliente, de comprobar si el objeto obtenido de manera dinámica desde la red es confiable y no malicioso; es decir, si el servidor que se encuentra a la escucha en "el otro lado" es la entidad que realmente ha generado el *proxy*, y no una tercera entidad maliciosa. Las librerías desarrolladas durante la elaboración de este proyecto no implementan esta funcionalidad, pero pueden ser necesarios estos mecanismos de protección en un entorno de distribución definitivo.

Para conseguir esta funcionalidad, los *proxies* y los servidores deberían implementar la interfaz `ProxyTrust` (en la práctica, la interfaz del servicio extendería `ProxyTrust`). A través del método *getProxyVerifier* el cliente obtendría por parte del servidor un objeto de tipo `TrustVerifier` creado durante la exportación del servicio y la creación del *proxy*. Este objeto contiene un método llamado *isTrustedObject* que acepta cualquier objeto remoto como parámetro y devuelve *true* si se comprueba que efectivamente el objeto *proxy* proporcionado coincide con aquél que se generó en el servidor en el momento de la exportación y publicación del servicio.

A.3. WRAPPER

El objeto remoto o *wrapper* es el segundo objeto que se registra en los servidores de *Lookup* para poner el servicio a disposición de los clientes de CAEAT. Su función es la de adaptar un objeto que sigue una semántica propia de las especificaciones *Jini / Apache River* (el *proxy*) a la semántica y estructura diseñada para los servicios utilizables por CAEAT (que siguen la semántica establecida por *JavaBeans*). Dado que el objeto remoto acaba encapsulando al *Proxy* y adaptándolo al entorno CAEAT, se ha denominado a éste con el nombre de *Wrapper*.

El *Wrapper* no es más que una clase poseedora de una estructura y propiedades idénticas a las de cualquier *bean* local comprensible por CAEAT (y por ende representable en el tapiz de la herramienta como un componente más). Sin embargo, las propiedades contenidas en el *Wrapper* son copias de las variables reales de proceso alojadas en el servidor, y los métodos proporcionados para el manejo de dichas variables acaban desembocando en llamadas remotas sobre el *proxy* contenido dentro del *Wrapper*. Esta clase, en definitiva, oculta a la herramienta CAEAT el hecho de que las llamadas a sus métodos se realizan de manera remota hacia un servidor; y al mismo tiempo gestiona las posibles excepciones que puedan ser lanzadas como consecuencia de errores en la comunicación.

Por todo lo explicado anteriormente, los métodos implementados por los objetos *Wrapper* pertenecen a dos grupos claramente diferenciados:

- Los métodos *getter* y *setter* del servicio, que permiten su uso y manejo
- Métodos propios de inicialización, gestión y mantenimiento del servicio

Los métodos del primer grupo son particulares y distintos para cada servicio, pero los del segundo resultan idénticos para todos los servicios. Es por ello que se ha definido una interfaz llamada `WrapperInterface` que obliga a todas las clases susceptibles de representar un objeto remoto de CAEAT a implementar la totalidad de estos métodos. Se presenta a continuación el código de dicha interfaz y se comenta brevemente la funcionalidad y el propósito de sus métodos:

```
package sener.publico.interfaces;

public interface WrapperInterface extends Serializable {

	public void EmbedProxy(Object p);
	public void setInitialProperties(Object[] props);
	public boolean isAlive();
	public long register(RemoteEventListener rel);
	public void unregister(String nombreInstancia, RemoteEventListener rel);
	public String getBeanInfoSearchPath();
	public void setServiceID(ServiceID id);
	public ServiceID getServiceID();
	public void setPropietario(String propietario);
	public String getPropietario();
	public String getNombreIcono();
	public void stopRemotely();
	public long getExpiration();
	public void setExpiration(long exp);
	public boolean resetService(long duration, RemoteEventListener owner);

}
```

Código B.3: Interfaz *WrapperInterface.class*

- ***EmbedProxy***: método utilizado en el momento de la obtención de los objetos *Proxy* y *Wrapper* para llevar a cabo la encapsulación del primero en el segundo.
- ***setInitialProperties***: en el momento de la inicialización de un servicio desde CAEAT, las variables del servidor no deben iniciarse a su valor por defecto, sino al valor que tenían en la versión local del servicio, previamente a su exportación. Este método proporciona al servidor dichos valores en un *array*.
- ***isAlive***: método utilizado por el mecanismo de control periódico de supervivencia de los servicios de CAEAT para comprobar si el servidor continúa activo (si lo está, se retorna *true*).
- ***register / unregister***: métodos empleados en el momento de la inserción y la eliminación del servicio representado por este *Wrapper* en el tapiz de CAEAT. Sirven para comunicar al servidor que un nuevo cliente desea recibir notificaciones remotas acerca del servicio (en el momento de su obtención e inserción en el tapiz) o que por el contrario desea darse de baja en dicho servicio y por consiguiente dejar de recibirlas (en el momento de la eliminación del tapiz de todas las instancias del servicio). En ambos casos se debe proporcionar el objeto `RemoteEventListener` que representa a la instancia de CAEAT que desea recibir los eventos.
- ***getBeanInfoSearchPath***: en la versión remota de los servicios, las clases `BeanInfo` y `Customizer` son cargadas de manera remota desde la máquina servidor. Es por ello que se debe proporcionar al cliente la ruta en la que se encuentran. Esta ruta es relativa respecto del directorio del servidor HTTP que sirve las clases en el lado del servidor. En modo local este paso no resultaba necesario puesto que las clases `BeanInfo` y `Customizer` ya se encontraban en el *Classpath* local junto con la propia clase del *bean*.
- ***setServiceID / getServiceID***: métodos que permiten consultar el `ServiceID` del servicio representado por este *Wrapper*, y establecerlo explícitamente en el momento de la publicación del servicio.
- ***setPropietario / getPropietario***: métodos para la consulta del propietario del servicio representado por este *Wrapper* y para su establecimiento en el momento de la publicación del mismo.
- ***getNombreIcono***: retorna el nombre del icono descriptivo del servicio, para poder cargarlo de manera remota fácilmente.

- ***stopRemotely***: envía al servidor una solicitud de detención del servicio. CAEAT garantiza que este método únicamente pueda ser invocado por el propietario del servicio.
- ***getExpiration / setExpiration***: métodos utilizados en el contexto del cambio de fecha de caducidad de un servicio. Sirven para consultar la fecha actual de caducidad y para establecer una fecha nueva.
- ***resetService***: solicita al servidor el reinicio del servicio. Se debe proporcionar el nuevo tiempo de vida y el `RemoteEventListener` que representa la instancia de CAEAT solicitante, puesto que todos los clientes excepto éste serán expulsados del servicio.

A continuación se presenta el código completo del *Wrapper* del servicio Contador de Letras:

```
package sener.publico.wrappers;

public class ContadorLetras implements Serializable, WrapperInterface {

	//Propiedades de gestión de la existencia del servicio:
	private static final long serialVersionUID = -108446915604721451L;
	private ServiceID id;
	private ContadorLetrasInterface p;
	private PropertyChangeSupport cambios;
	private String propietario;
	private final String nombreIcono = "ContadorLetras.png";

	//Propiedades del bean:
	private String mensaje;
	private int letras;
	private boolean cuenta;
	private boolean contajeAutomatico;

	public ContadorLetras() {
		mensaje = "Cuenta las letras contenidas en este mensaje";
		cuenta = false;
		contajeAutomatico = false;
		letras = 0;
		cambios = new PropertyChangeSupport(this);
	}

	public void EmbedProxy(Object p) {
		this.p = (ContadorLetrasInterface)p;
		initialUpdate();
	}

	public boolean isAlive() {
		try {
			return p.isAlive();
		} catch (RemoteException e) {
			e.printStackTrace();
		}
		return false;
	}

	public ServiceID getServiceID() {
		return id;
	}

	public void setServiceID(ServiceID id) {
		this.id = id;
	}

	public long getExpiration() {
		try {
			return p.getExpiration();
		} catch(RemoteException e) {
			e.printStackTrace();
			return 0;
		}
	}
```

```
	public void setExpiration(long exp) {
		try {
			p.setExpiration(id,exp);
		} catch(RemoteException e) { e.printStackTrace(); }
	}

	public String getNombreIcono() {
		return nombreIcono;
	}

	public void setPropietario(String propietario) {
		this.propietario = propietario;
	}

	public String getPropietario() {
		return propietario;
	}

	public boolean resetService(long duration, RemoteEventListener propietario) {
		boolean b = false;
		try {
			b = p.resetService(id, duration, propietario);
		} catch(RemoteException e) {
			e.printStackTrace();
			return false;
		}
		updateAllProperties();
		return b;
	}

	public String getMensaje() {
		String oldMensaje = mensaje;
		try {
			if(!oldMensaje.equals(p.getMensaje())) {
				mensaje = p.getMensaje();
				cambios.firePropertyChange("mensaje", oldMensaje, mensaje);
			}
		} catch(RemoteException e) { e.printStackTrace(); }
		return mensaje;
	}

	public void setMensaje(String nuevomensaje) {
		String oldMensaje = this.mensaje;
		try {
			if(!oldMensaje.equals(nuevomensaje)) {
				this.mensaje = nuevomensaje;
				p.setMensaje(nuevomensaje);
				cambios.firePropertyChange("mensaje", oldMensaje, mensaje);
				realizaCuenta(contajeAutomatico);
			}
		} catch(RemoteException e) { e.printStackTrace(); }
	}

	public int getLetras() {
		int oldLetras = letras;
		try {
			if(oldLetras != p.getLetras()) {
				letras = p.getLetras();
				cambios.firePropertyChange("letras", oldLetras, letras);
			}
		} catch(RemoteException e) { e.printStackTrace(); }
		return letras;
	}

	public boolean isContajeAutomatico() {
		boolean oldContajeAutomatico = contajeAutomatico;
		try {
			if(oldContajeAutomatico != p.isContajeAutomatico()) {
				contajeAutomatico = p.isContajeAutomatico();
				cambios.firePropertyChange("contajeAutomatico",
				oldContajeAutomatico, contajeAutomatico);
			}
		} catch(RemoteException e) { e.printStackTrace(); }
		return contajeAutomatico;
	}
```

```
	public void setContajeAutomatico(boolean contajeAutomatico) {
		boolean old = this.contajeAutomatico;
		try {
			if(old != contajeAutomatico) {
				this.contajeAutomatico = contajeAutomatico;
				p.setContajeAutomatico(contajeAutomatico);
				cambios.firePropertyChange("contajeAutomatico", old,
				contajeAutomatico);
			}
		} catch(RemoteException e) { e.printStackTrace(); }
	}

	public boolean isCuenta() {
		boolean oldCuenta = cuenta;
		try {
			if(oldCuenta != p.isCuenta()) {
				cuenta = p.isCuenta();
				cambios.firePropertyChange("cuenta", oldCuenta, cuenta);
			}
		} catch(RemoteException e) { e.printStackTrace(); }
		return cuenta;
	}

	public void realizaCuenta(boolean cuenta) {
		int nuevoLetras = 0;
		int oldLetras = this.letras;
		if(cuenta) {
			StringBuffer buff = new StringBuffer(mensaje);
			for(int i=0 ; i<buff.length() ; i++) {
				char c = buff.charAt(i);
				if((c >= 65 && c <= 90) || (c >= 97 && c <= 122))
					nuevoLetras++;
			}
			try {
				p.realizaCuenta(cuenta);
			} catch(RemoteException e) { e.printStackTrace(); }
			cambios.firePropertyChange("cuenta", false, cuenta);
			cambios.firePropertyChange("cuenta", true, false);
			cambios.firePropertyChange("letras", oldLetras, nuevoLetras);
		}
		this.cuenta = false;
		this.letras = nuevoLetras;
	}

	public void addPropertyChangeListener(PropertyChangeListener l) {
		cambios.addPropertyChangeListener(l);
	}

	public void removePropertyChangeListener(PropertyChangeListener l) {
		cambios.removePropertyChangeListener(l);
	}

	public PropertyChangeListener[] getPropertyChangeListeners() {
		return cambios.getPropertyChangeListeners();
	}

	public PropertyChangeListener[] getPropertyChangeListeners(String propertyName) {
		return cambios.getPropertyChangeListeners(propertyName);
	}

	public String toString() {
		return "Contador de letras [mensaje = " + mensaje + " // Letras = " + letras
			+ "]";
	}

	public long register(RemoteEventListener rel) {
		long eventSource = 0;
		try {
			eventSource = p.register(rel);
		} catch(RemoteException e) { e.printStackTrace(); }

		return eventSource;
	}
```

```
        public void unregister(String nombreInstancia, RemoteEventListener rel) {
                try {
                        p.unregister(nombreInstancia, rel);
                } catch(RemoteException e) { e.printStackTrace(); }
        }

        private void initialUpdate() {
                try {
                        mensaje = p.getMensaje();
                        cuenta = p.isCuenta();
                        contajeAutomatico = p.isContajeAutomatico();
                        letras = p.getLetras();
                } catch(RemoteException e) { e.printStackTrace(); }
        }

        private void updateAllProperties() {
                getMensaje();
                getLetras();
                isContajeAutomatico();
                isCuenta();
        }

        public void setInitialProperties(Object[] props) {
                try {
                        p.setInitialProperties(props);
                } catch (RemoteException e) { e.printStackTrace(); }
        }

        public String getBeanInfoSearchPath() {
                return "sener.pruebas.analisis";
        }

        public void stopRemotely() {
                try {
                        p.stopRemotely(id);
                } catch (RemoteException e) { e.printStackTrace(); }
        }

}
```

Código B.4: Implementación completa de la clase *ContadorLetrasWrapper.class*

Cabe destacar las siguientes características del código recién presentado:

- Las propiedades del servicio (`mensaje`, `letras`, `contajeAutomatico`, etc.) son copias actualizadas de las propiedades reales que se alojan en el servidor, y están claramente diferenciadas de las propiedades de gestión y mantenimiento del servicio (`id`, `cambios`, `nombreIcono`, `propietario`, etc.).
- Como cualquier *bean* local, el *Wrapper* contiene una lista de *listeners* a los cuales es posible lanzar eventos de cambios en el valor de las propiedades (`PropertyChangeEvent`). Este hecho permite que los servicios remotos puedan ser conectados de manera transparente con cualquier otro servicio (ya sea local o remoto) ubicado en el tapiz de CAEAT. Un `PropertyChangeEvent` es lanzado cada vez que se lleva a cabo un cambio sobre una propiedad del servicio.
- La llamada a cualquiera de los métodos *getter* del servicio es aprovechada para actualizar el valor de la copia local de la propiedad en el caso de que éste se encontrase desactualizado.
- Es posible que bajo ciertas circunstancias se produzcan cadenas de llamadas a los métodos del servicio. Por ejemplo, el establecimiento de un nuevo mensaje provoca el contaje automático de las letras contenidas en éste si la propiedad `contajeAutomatico` es igual a *true*. Estas cadenas son reproducidas en el servidor y los eventos pertinentes son lanzados para garantizar que todos los clientes poseen una versión actualizada de las variables.
- Se proporcionan los métodos privados *initialUpdate* y *updateAllProperties* para realizar una actualización de todas las propiedades (actualización inicial y en caliente, respectivamente).

B.4. SERVIDOR

La clase *Servidor* aloja las variables reales de proceso, y en ella tiene lugar el procesado real del servicio. Únicamente existe una instancia de esta clase en toda la red, tratándose del objeto que se exporta en la máquina proveedora del servicio para la recepción de llamadas remotas por parte de los clientes. Esta clase ofrece implementación tanto para todos los métodos propios del servicio (*getters* y *setters*) como para todos los métodos de gestión y mantenimiento de éste.

Entre las responsabilidades principales de esta clase se encuentra la de mantener una lista actualizada con los clientes que en cada momento están suscritos al servicio, para poder enviar a todos ellos las notificaciones de cambio de valor en las propiedades y eventos similares. Debido a que la problemática del acceso concurrente a las propiedades en un entorno distribuido como el propuesto por *Jini / Apache River* tiene lugar en esta clase, también proporciona protección contra dicho acceso mediante la utilización de monitores.

A continuación se muestra el código completo de la clase *Servidor* del servicio Contador de Letras, y se comentan sus particularidades y métodos más destacables:

```
package sener.pruebas.analisis.servicios;

public class ContadorLetrasServer implements Serializable, ContadorLetrasInterface {

	//Propiedades de gestión del servicio:
	final static long serialVersionUID = 1776300912334522111L;
	private boolean alive;
	private CopyOnWriteArrayList<RemoteEventListener> CAEATListeners;
	private Long eventSource;
	private long eventSequence;
	private MarshalledObject<SenetBeansNotification> handback;
	private ActionListener CSM;
	private long expiration;

	//Propiedades del bean:
	private String mensaje;
	private boolean cuenta;
	private boolean contajeAutomatico;
	private int letras;

	//Monitores de protección de las variables de proceso:
	private SenetBeansMonitor monMensaje;
	private SenetBeansMonitor monCuenta;
	private SenetBeansMonitor monContajeAutomatico;
	private SenetBeansMonitor monLetras;

	public ContadorLetrasServer() {
		mensaje = "Cuenta las letras contenidas en este mensaje";
		cuenta = false;
		contajeAutomatico = false;
		letras = 0;
		monMensaje = new SenetBeansMonitor();
		monCuenta = new SenetBeansMonitor();
		monContajeAutomatico = new SenetBeansMonitor();
		monLetras = new SenetBeansMonitor();
		Random r = new Random();
		alive = true;
		CAEATListeners = new CopyOnWriteArrayList<RemoteEventListener>();
		do {
			eventSource = r.nextLong();
		} while(eventSource == -1);
		eventSequence = 1;
	}
```

```
	private void initialState() {
		mensaje = "Cuenta las letras contenidas en este mensaje";
		cuenta = false;
		contajeAutomatico = false;
		letras = 0;
		monMensaje = new SenetBeansMonitor();
		monCuenta = new SenetBeansMonitor();
		monContajeAutomatico = new SenetBeansMonitor();
		monLetras = new SenetBeansMonitor();
	}

	public long register(RemoteEventListener CAEATInstance) {
		if(!CAEATListeners.contains(CAEATInstance)) {
			CAEATListeners.add(CAEATInstance);
		}
		return eventSource;
	}

	public void unregister(String nombreInstancia, RemoteEventListener rel) {
		lanzaEvento(0,nombreInstancia,null);
		CAEATListeners.remove(rel);
	}

	public void finish() {
		lanzaEvento(2,null,null);
	}

	public boolean resetService(ServiceID id, long duration,
		RemoteEventListener owner) {
		CAEATListeners.remove(owner);
		finish();
		register(owner);
		initialState();
		setExpiration(id, duration);
		return true;
	}

	public void setInitialProperties(Object[] props) {
		if (props != null) {
			this.contajeAutomatico = (Boolean)props[0];
			this.cuenta = (Boolean)props[1];
			this.letras = (Integer)props[2];
			this.mensaje = (String)props[3];
		}
	}

	public void setCSMListener(ActionListener CSM) {
		this.CSM = CSM;
	}

	public boolean isAlive() {
		return alive;
	}

	public void setAlive(boolean alive) {
		this.alive = alive;
	}

	public long getExpiration() {
		return expiration;
	}

	public void setExpiration(ServiceID id, long exp) {
		if(expiration > 0) {
			SenetBeansNotification not = new
			SenetBeansNotification(eventSource,"changeDate",
				new Object[]{ new Long(exp)});
			CSM.actionPerformed(new
			ActionEvent(not,ActionEvent.ACTION_PERFORMED,id.toString()));
		}
		this.expiration = exp;
	}
```

```
	public String getMensaje() {
		String mensaje;
		monMensaje.Leer();
		mensaje = this.mensaje;
		monMensaje.salirLectura();
		return mensaje;
	}

	public void setMensaje(String mensaje) {
		monMensaje.Escribir();
		this.mensaje = mensaje;
		monMensaje.salirEscritura();
		lanzaEvento(1,"mensaje",null);
	}

	public int getLetras() {
		int letras;
		monLetras.Leer();
		letras = this.letras;
		monLetras.salirLectura();
		return letras;
	}

	public boolean isContajeAutomatico() {
		boolean contajeAutomatico;
		monContajeAutomatico.Leer();
		contajeAutomatico = this.contajeAutomatico;
		monContajeAutomatico.salirLectura();
		return contajeAutomatico;
	}

	public void setContajeAutomatico(boolean contajeAutomatico) {
		monContajeAutomatico.Escribir();
		this.contajeAutomatico = contajeAutomatico;
		monContajeAutomatico.salirEscritura();
		lanzaEvento(1,"contajeAutomatico",null);
	}

	public boolean isCuenta() {
		boolean cuenta;
		monCuenta.Leer();
		cuenta = this.cuenta;
		monCuenta.salirLectura();
		return cuenta;
	}

	public void realizaCuenta(boolean cuenta) {
		int nuevoLetras = 0;
		monMensaje.Leer();
		monCuenta.Escribir();
		monLetras.Escribir();
		if(cuenta) {
			StringBuffer buff = new StringBuffer(mensaje);
			for(int i=0 ; i<buff.length() ; i++) {
				char c = buff.charAt(i);
				if( (c >= 65 && c <= 90) || (c >= 97 && c <= 122))
					nuevoLetras++;
			}
		}
		this.cuenta = false;
		this.letras = nuevoLetras;
		monLetras.salirEscritura();
		monCuenta.salirEscritura();
		monMensaje.salirLectura();
		lanzaEvento(1,"letras",null);
	}

	private void notifyCAEAT(RemoteEvent ev) {
		for(RemoteEventListener caeat : CAEATListeners) {
			new SenetBeansNotifier(caeat,ev,CAEATListeners);
		}
	}
```

```
        private void lanzaEvento(int tipoEvento, String info, Object[] objetos) {
                // 0 = Petición de desregistro
                // 1 = Actualización de alguna propiedad
                // 2 = Notificación de detención o caducidad de ese servicio
                // 3 = Cambio estructural en una agregación remota
                try {
                        handback = new MarshalledObject<SenetBeansNotification>(new
                        SenetBeansNotification(eventSource,info,objetos));
                } catch(IOException e) {e.printStackTrace(); }
                RemoteEvent ev = new
                RemoteEvent(this.getClass().getName(),tipoEvento,eventSequence,handback);
                eventSequence = ((eventSequence + 1) % Long.MAX_VALUE);
                notifyCAEAT(ev);
        }

        public void stopRemotely(ServiceID id) throws RemoteException {
                CSM.actionPerformed(new
                ActionEvent(this,ActionEvent.ACTION_PERFORMED,id.toString()));
        }

        public String toString() {
                return "Contador de letras [mensaje = " + mensaje + " // Letras = " +
                         letras + "]";
        }
}
```

Código B.5: Implementación completa de la clase *ContadorLetrasServer.class*

En primer lugar, se debe hacer notar la utilización de un monitor por cada propiedad a proteger. Las razones de esta decisión fueron ampliamente expuestas en el capítulo 3.5, dedicado a la problemática de la concurrencia en llamadas remotas. En operaciones que implican el manejo de más de una variable de proceso, como el método *realizaCuenta*, se debe obtener el monitor de todas ellas para poder operar.

La clase *Servidor* hace uso de dos nuevas clases adicionales no expuestas hasta el momento: `SenetBeansNotification` y `SenetBeansNotifier`. La primera de ellas es un simple contenedor de objetos útil a la hora de realizar las notificaciones de eventos a los clientes. La segunda de ellas ayuda a realizar el lanzamiento de las notificaciones a los clientes bajo una disciplina *multithread*. A continuación se presentan los detalles de implementación de ambas clases:

```
package sener.publico.gestion;

public class SenetBeansNotifier extends Thread {

        private RemoteEventListener client;
        private RemoteEvent ev;
        private CopyOnWriteArrayList<RemoteEventListener> listeners;

        public SenetBeansNotifier(RemoteEventListener client, RemoteEvent ev,
                CopyOnWriteArrayList<RemoteEventListener> listeners) {
                this.client = client;
                this.ev = ev;
                this.listeners = listeners;
                start();
        }

        public void run() {
                try {
                        client.notify(ev);
                } catch(RemoteException e) {
                        listeners.remove(client);
                }
                catch(UnknownEventException e) {
                        listeners.remove(client);
                }
        }
}
```

Código B.6: Implementación de la clase *SenetBeansNotifier.class*

```
package sener.publico.gestion;

public class SenetBeansNotification implements Serializable {

	private static final long serialVersionUID = 5428563200145478541L;
	public long eventSource;
	public String info;
	public Object[] objetos;

	public SenetBeansNotification(long eventSource, String info, Object[] objetos) {
		this.eventSource = eventSource;
		this.info = info;
		this.objetos = objetos;
	}

}
```

Código B.7: Implementación de la clase *SenetBeansNotification.class*

Ambas clases son conocidas por la plataforma CAEAT ya que pertenecen a su estructura natural de clases, y deben ser incluidas y empleadas en una librería de componentes apta para ser utilizable con CAEAT. Pese a que pueda parecer que dos clases tan sencillas como éstas (un contenedor y una clase que ejecuta una tarea sencilla) pueden ser incluidas dentro de la clase Servidor como sub-clases o como clases anónimas, esto rompería la estructura de clases canónica de publicación de servicios, formada únicamente por las cinco clases presentadas en este anexo, dificultando el proceso de publicación y lanzamiento de servicios.

Las operaciones de lectura, por lo general, se limitan a retornar el valor actual de la propiedad a leer. Sin embargo, las operaciones de escritura provocan un cambio en el valor de una propiedad, que debe ser inmediatamente notificado a todos los clientes del servicio mediante el método *lanzaEvento*. Este método sirve para notificar a los clientes actuales de la ocurrencia de un evento significativo en el servicio, que no necesariamente debe ser un evento de cambio de valor en las propiedades, sino que admite los siguientes tipos:

- Notificación de desregistro de un cliente. Al cancelar un cliente su suscripción al servicio, es necesario enviar una notificación de regreso a dicho cliente.
- Cambio en el valor de alguna propiedad del servicio.
- Notificación de finalización del servicio (ya sea por detención manual por parte de su propietario o por caducidad).
- Notificación de cambio estructural en la agregación (sólo para el caso de agregaciones remotas).

A continuación se comenta la funcionalidad del resto de métodos de gestión de la vida del servicio implementados por la clase *Servidor*:

- ***isAlive / setAlive***: el primer método retorna el valor *true* para informar al cliente de que el servicio continúa activo. El segundo sirve para poner de manera artificial al servicio en un estado de inactividad a ojos de los clientes (el método *isAlive* retornará *false* si así se ha establecido). Esta característica es útil en el caso de la des-publicación de los servicios, en la que éstos simulan estar inactivos para cualquier cliente que no se esté ejecutando desde la misma máquina en la que reside el servicio.
- ***register / unregister***: métodos que actualizan la lista de clientes que se encuentran suscritos al servicio. En el caso de *register*, se devuelve un número identificador del servicio para que el cliente pueda identificar el generador de los eventos recibidos.
- ***finish***: lanza a los clientes el evento de finalización del servicio.
- ***setInitialProperties***: establece los valores iniciales de las variables de proceso del servidor (usado únicamente en el momento de la creación de dicho servidor).

- ***setCSMListener***: la clase *Servidor* es capaz de enviar notificaciones a la instancia de *CAEAT Service Manager* que la está gestionando. Para ello, en el momento de la inicialización del servicio, se registra CSM como *listener* del *Servidor* en cuestión.
- ***stopRemotely***: notifica a *CAEAT Service Manager* que el servicio debe ser detenido. CSM se encargará de detener adecuadamente el servicio, des-exportando el objeto *Servidor* y des-publicando sus objetos públicos de los servidores de *Lookup*.
- ***getExpiration / setExpiration***: permiten consultar la fecha de caducidad actual del servicio y establecer una nueva, respectivamente. En el caso del segundo método, la notificación se redirecciona a *CAEAT Service Manager*, puesto que es esta entidad la encargada de gestionar las fechas de caducidad de los servicios.
- ***resetService***: envía un evento de finalización del servicio a todos los clientes excepto al realizador de la solicitud de reinicio. A continuación devuelve a su estado inicial todas las variables de proceso y establece la nueva fecha de caducidad recibida.

B.5. PUBLICADOR

La clase *Publicador* ayuda a realizar abstracción, en el momento de la publicación de los servicios, entre la naturaleza de la plataforma publicadora (CAEAT en este caso, pero podría tratarse de otra plataforma en el futuro) y la naturaleza y diseño de los objetos manejados en la operación (*proxy*, *wrapper*, etc). La clase *Publicador* debe definir de manera precisa las operaciones que es capaz de llevar a cabo para conseguir la publicación y mantenimiento de un servicio, de manera que la plataforma publicadora únicamente deba invocar los métodos adecuados para conseguir tal fin.

Para ello, se ha definido una interfaz llamada `PublicadorInterface`, que deberá ser implementada por la clase *Publicador* y conocida por la plataforma publicadora (CAEAT y similares). Dicha interfaz se presenta a continuación:

```
package sener.publico.interfaces;

public interface PublicadorInterface {

	public String getNombre();
	public String getTipo();
	public URL getIcono();
	public ServiceID getServiceID();
	public ArrayList<Lease> getLeases();
	Object[] publicarServicio(Object simpleService,
		ServiceRegistrar[] servers, String[] groups,ActionListener CSMListener);
	Object[] publicarBean (Object beanServer, Object beanWrapper,
		ServiceRegistrar[] servers, String[] groups, ActionListener CSMListener);
	boolean unexport();
	boolean unpublish();
	public void setServerSettings(Name tipoServicio, int puerto, String propietario);
	public String getPropietario();
	public boolean isPublicado();
	boolean isViral();
	void setViral(boolean viral);
	ArrayList<ServiceRegistrar> getServers();
	String[] getGroups();
	PublicadorInterface rePublicarBean(Object beanWrapper, long lifeTime,
			ServiceRegistrar[] servers, String[] groups);
	PublicadorInterface rePublicarServicio(long newLifeTime,
		ServiceRegistrar[] servers, String[] groups);
	void expandirServicio(ArrayList<ServiceRegistrar> newServers,
		ArrayList<ServiceRegistrar> discardedServers);
}
```

Código B.8: Interfaz *PublicadorInterface.class*

Como se ha comentado, la interfaz define una serie de métodos que establecen claramente las operaciones de publicación y gestión que una plataforma de edición y publicación es capaz de utilizar, así como los parámetros que dichos métodos deben recibir por parte de la plataforma. Son responsabilidad del creador de la librería de componentes los detalles de implementación de dichos métodos y las estrategias adoptadas para el correcto manejo y publicación de los objetos.

A continuación se presenta el código completo de la clase *Publicador* del servicio Contador de Letras, y se comenta brevemente la funcionalidad de cada uno de sus métodos:

```
package sener.pruebas.analisis.servicios;

public class ContadorLetrasPubli implements PublicadorInterface, ProxyAccessor, Serializable
{
	private static final long serialVersionUID = -1995145870332101481L;
	private ContadorLetrasInterface inter;
	private ContadorLetrasInterface proxy;
	private transient Object beanWrapper;
	private ServiceID serviceIDProxy;
	private ArrayList<Lease> leases;
	private int puerto;
	private boolean isPublicado;
	private String propietario;
	private Name tipoServicio;
	private transient BasicJeriExporter bje;
	private boolean viral;
	private String[] gruposRegistro;
	private transient ArrayList<ServiceRegistrar> lookupServers;

	public ContadorLetrasPubli() {
		leases = new ArrayList<Lease>();
		puerto = -1;
		isPublicado = false;
	}

	public void setServerSettings(Name tipoServicio, int puerto, String propietario) {
		this.tipoServicio = tipoServicio;
		this.puerto = puerto;
		this.propietario = propietario;
	}

	private BasicJeriExporter getExporter() {
		bje = new BasicJeriExporter(TcpServerEndpoint.getInstance(puerto), new
			BasicILFactory(), true, true);
		return bje;
	}

	public boolean isPublicado() {
		return isPublicado;
	}

	public boolean isViral() {
		return viral;
	}

	public void setViral(boolean viral) {
		this.viral = viral;
	}

	public ArrayList<ServiceRegistrar> getServers() {
		return lookupServers;
	}

	public String[] getGroups() {
		return gruposRegistro;
	}
```

```
	private void RegistraLUS(Object pub, Entry[] datos, ServiceRegistrar[] servers,
	boolean keepID) throws IOException {
		ServiceItem item;
		if(keepID && pub.getClass().getName().contains("Proxy"))
			item = new ServiceItem(serviceIDProxy,pub,datos);
		else
			item = new ServiceItem(null,pub,datos);

		for(ServiceRegistrar sr : servers) {
			ServiceRegistration registration = null;
			try {
				registration = sr.register(item,Lease.FOREVER);
				leases.add(registration.getLease());
			} catch (RemoteException e) {
				return;
			}
			if(item.serviceID == null &&
			pub.getClass().getName().contains("Proxy")) {
				item.serviceID = registration.getServiceID();
				serviceIDProxy = item.serviceID;
			}
		}
	}

	public Object[] publicarBean(Object beanServer, Object beanWrapper,
		ServiceRegistrar[] servers, String[] groups, ActionListener CSMListener) {
		if (servers.length == 0) {
			return null;
		}
		lookupServers = new ArrayList<ServiceRegistrar>(Arrays.asList(servers));
		gruposRegistro = groups;
		this.beanWrapper = beanWrapper;
		bje = getExporter();
		((ContadorLetrasServer)beanServer).setCSMListener(CSMListener);
		try {
		inter = (ContadorLetrasInterface)bje.export(
			(ContadorLetrasServer)beanServer);
		} catch (ExportException e) { e.printStackTrace(); }

		proxy = ContadorLetrasProxy.crear(inter);
		try {
			RegistraLUS(proxy,new Entry[]{tipoServicio},servers,false);
			ServiceInfo siWrap = new
			ServiceInfo(null,null,null,null,null,serviceIDProxy.toString());
			ServiceInfo[] siWrapArray = new ServiceInfo[]{siWrap};
			((WrapperInterface)beanWrapper).setServiceID(serviceIDProxy);
			RegistraLUS(beanWrapper,siWrapArray,servers,false);
		} catch(IOException e) { e.printStackTrace(); }
		isPublicado = true;
		return new Object[]{proxy, serviceIDProxy, this};
	}

	public PublicadorInterface rePublicarBean(Object beanWrapper, long newLifeTime,
			ServiceRegistrar[] servers, String[] groups) {
		if (servers.length == 0) {
			return null;
		}
		lookupServers = new ArrayList<ServiceRegistrar>(Arrays.asList(servers));
		gruposRegistro = groups;
		this.beanWrapper = beanWrapper;
		try {
			RegistraLUS(proxy,new Entry[]{tipoServicio},servers,true);
			ServiceInfo siWrap = new
			ServiceInfo(null,null,null,null,null,serviceIDProxy.toString());
			ServiceInfo[] siWrapArray = new ServiceInfo[]{siWrap};
			((WrapperInterface)beanWrapper).setServiceID(serviceIDProxy);
			RegistraLUS(beanWrapper,siWrapArray,servers,true);
		} catch(IOException e) {
			e.printStackTrace();
			return null;
		}
```

```
		try {
			proxy.setAlive(true);
			proxy.setExpiration(serviceIDProxy,
				System.currentTimeMillis() + newLifeTime);
		} catch(RemoteException e) {
			e.printStackTrace();
			return null;
		}
		isPublicado = true;
		return this;
	}

	public PublicadorInterface rePublicarServicio(long newLifeTime,
		ServiceRegistrar[] servers, String[] groups) {
		//Únicamente implementado por los publicadores de servicios
		return null;
	}

	public ArrayList<Lease> getLeases() {
		return leases;
	}

	public boolean unexport() {
		try {
			inter.finish();
		} catch(RemoteException e) {
			e.printStackTrace();
			return false;
		}
		return bje.unexport(true);
	}

	public boolean unpublish() {
		try {
			inter.setAlive(false);
			inter.finish();
		} catch(RemoteException e) {
			e.printStackTrace();
			return false;
		}
		isPublicado = false;
		leases.clear();
		return true;
	}

	public String getNombre() {
		return "Contador de Letras";
	}

	public String getTipo() {
		return "Servicio remoto simple";
	}

	public ServiceID getServiceID() {
		return serviceIDProxy;
	}

	public URL getIcono() {
		//ContadorLetras.png
		URL u = getClass().getResource("/sener/pruebas/imagenes/ContadorLetras.png");
		if(u != null)
			return u;
		try {
			u = new URL(RMIClassLoader.getClassAnnotation(getClass()) +
			"imagenes/iconos/ContadorLetras.png");
		} catch(MalformedURLException e) {
			e.printStackTrace();
			return null;
		}
		return u;
	}

	public Object getProxy() {
		return proxy;
	}
```

```
        public Object[] publicarServicio(Object simpleService, ServiceRegistrar[] servers,
                String[] groups, ActionListener CSMListener) {
                //Método implementado únicamente por los publicadores de librerías u otros
                //servicios sencillos
                return null;
        }

        public String getPropietario() {
                return propietario;
        }

        public void expandirServicio(ArrayList<ServiceRegistrar> newServers,
                        ArrayList<ServiceRegistrar> discardedServers) {
                lookupServers.removeAll(discardedServers);
                if(newServers.size() > 0) {
                ServiceRegistrar[] servers = newServers.toArray(new ServiceRegistrar[]{});
                        try {
                                RegistraLUS(proxy,new Entry[]{tipoServicio},servers,true);
                                ServiceInfo siWrap = new
                                ServiceInfo(null,null,null,null,null,
                                        serviceIDProxy.toString());
                                ServiceInfo[] siWrapArray = new ServiceInfo[]{siWrap};
                                ((WrapperInterface)beanWrapper).setServiceID(serviceIDProxy);
                                RegistraLUS(beanWrapper,siWrapArray,servers,true);
                        } catch(IOException e) {
                                e.printStackTrace();
                        }
                }
                lookupServers.addAll(newServers);
        }
}
```

Código B.9: Implementación completa de la clase *ContadorLetrasPubli.class*

La clase *Publicador* contiene atributos que resultan útiles en el manejo del servicio publicado, como por ejemplo:

- Los grupos de registro en los cuales se encuentra publicado el servicio, y los `ServiceRegistrar` (objetos descriptores de los servidores LUS) que definen el conjunto de servidores en los que se ha publicado.
- La lista de *Leases* o concesiones recibidas por parte de los servidores LUS, para que *CAEAT Service Manager* pueda renovarlos o cancelarlos según corresponda.
- Una copia de los objetos *Proxy* y *Wrapper* del servicio, junto con el `ServiceID` del mismo.
- El objeto `BasicJeriExporter` usado para exportar el *Servidor*, juntamente con el puerto (escogido al azar) en el cual dicho servidor permanecerá a la escucha de llamadas remotas por parte de los clientes.
- Información básica sobre el servicio: su tipo, su propietario, valores de tipo *boolean* que indican si éste es viral o si se encuentra despublicado, etc.

Los métodos que se comentan a continuación se enumeran en el mismo orden en el cual se presentaron en la interfaz `PublicadorInterface` (código B.8):

- ***getNombre / getTipo / getIcono / getServiceID***: métodos que proporcionan la información que indica su nombre, que es utilizada para diversas finalidades. En el caso del nombre y el tipo, se trata de cadenas de caracteres introducidos en el código del *Publicador* durante su desarrollo. En el caso del icono, se devuelve una URL que apunta a la localización del servidor HTTP en la que se alojan los recursos accesibles desde el exterior. El `ServiceID` del servicio queda automáticamente establecido en el momento de su publicación.
- ***getLeases***: devuelve la colección de todos los objetos `Lease` que representan las concesiones de todos los servidores de *Lookup* en los cuales se han registrado los objetos públicos del servicio (tanto *Proxy* como *Wrapper*)

- ***publicarServicio / publicarBean***: métodos que llevan a cabo el proceso de comunicación con los servidores de *Lookup* y la publicación de los objetos públicos del servicio. La principal diferencia entre ellos dos radica en el hecho de que *publicarBean* será implementado por los publicadores de *beans*, que deben registrar dos objetos en los servidores LUS (*Proxy* y *Wrapper*); mientras que *publicarServicio* será implementado por los publicadores de servicios (ya sean librerías, repositorios de imágenes, etc.,) y éstos únicamente deben registrar el objeto *Proxy* en los servidores.

  Ambos métodos reciben el conjunto de objetos `ServiceRegistrar` que representan los servidores de *Lookup* en los que se desea registrar el servicio (la tarea de búsqueda de dichos servicios recae sobre la plataforma publicadora). También se proporcionan los nombres de los grupos de *Lookup* de destino y una referencia a *CAEAT Service Manager* en forma de `ActionListener`, para que la clase *Servidor* pueda lanzarle eventos.

- ***unexport / unpublish***: usados en el momento de la finalización y la des-publicación de un servicio respectivamente. Ambos envían a la clase *Server* la solicitud de finalización de servicio para que ésta la pueda redireccionar a los clientes. El primero (*unexport*) fuerza a que el objeto *Server* deje de estar exportado, y por tanto se dejarán de atender llamadas por parte de los clientes. El segundo limpia la lista de objetos `Lease` de los servidores de *Lookup* y solicita a la clase *Servidor* que devuelva *false* ante un sondeo de supervivencia.

- ***setServerSettings***: método usado en el momento de la publicación del servicio para establecer algunos parámetros básicos que deben haber sido obtenidos por la plataforma publicadora: su tipo, su propietario y el puerto en el cual el objeto *Servidor* restará a la escucha de llamadas remotas provinentes de los clientes.

- ***getPropietario / isPublicado***: métodos que retornan la información indicada por su nombre.

- ***isViral / setViral***: consulta y establece, respectivamente, el estado de viralidad del servicio. Este parámetro es consultado por el *Thread* que se encarga de la expansión de los servicios virales.

- ***getServers / getGroups***: retornan, respectivamente, el nombre de los grupos de *Lookup* en los cuales se encuentra publicado el servicio y la colección de objetos `ServiceRegistrar` que representan los servidores LUS en los que el objeto está actualmente publicado. Estos métodos son utilizados por el *Thread* expansor de los servicios virales (únicamente en el caso de que el servicio efectivamente lo sea).

- ***rePublicarBean / rePublicarServicio***: métodos utilizados para re-publicar *beans* y servicios que habían sido previamente des-publicados. El proceso de re-publicación es ligeramente distinto al de la primera publicación porque los objetos públicos ya existen y únicamente se deben registrar de nuevo en los servidores de *Lookup* que se proporcionan como parámetro.

  Además, en el proceso de re-publicación se debe garantizar la utilización del mismo `ServiceID` del cual el servicio ya gozaba anteriormente a su des-publicación. No es necesario exportar el objeto *Servidor* ya que en ningún momento se habrá des-exportado, pero sí se le debe proporcionar la nueva fecha de caducidad y notificarle que ya puede volver a contestar *true* ante las solicitudes de estado de supervivencia.

- ***expandirServicio***: método únicamente invocado por el *Thread* de expansión de los servicios virales en caso de que el servicio efectivamente lo sea. Dicho *Thread* habrá realizado el cribado de servidores *Lookup* y habrá determinado si existen nuevos servidores en los que el servicio deba registrarse, o si han desaparecido algunos de los que se estaban manteniendo como referencia. Mediante este método se solicita al servicio su registro en los servidores LUS nuevos, o la eliminación de la referencia a los desaparecidos (ambas listas se obtienen externamente y se proporcionan como parámetro).

C. PROTECCIÓN DE LOS ATRIBUTOS DE LOS SERVICIOS

En el capítulo 3 se expuso la necesidad de protección de las variables de proceso del servidor ante el acceso concurrente por parte de los clientes. Se propuso como solución el uso de alguno de los clásicos monitores aplicados a la problemática de lectores / escritores. Se mostró la manera de proteger las variables en la clase *Server* que las contiene, pero no se concretó la implementación de ningún monitor en concreto.

Como la mayoría de monitores que se usan en los escenarios de tipo lectores / escritores, un monitor apto para ser usado con los servicios desarrollados para CAEAT debe implementar cuatro métodos: `Leer()`, `salirLectura()`, `Escribir()` y `salirEscritura()`. Internamente, el monitor permitirá a los *threads* el acceso a los recursos atendiendo a las variables de condición que se hayan establecido (reglas que se deben cumplir para que un *Thread* pueda tomar control del monitor y leer o escribir el recurso).

Este anexo presenta y comenta el código de `SenetBeansMonitor`, el monitor que se ha implementado en la adaptación a remoto de los servicios desarrollados para CAEAT.

C.1. POLÍTICA DE FUNCIONAMIENTO

Un monitor aplicado a la resolución de un escenario de lectores / escritores como el que se tiene en el presente proyecto puede implementar distintas disciplinas en lo que se refiere al entrelazado de ejecuciones de lectores y escritores. La solución más sencilla, por ejemplo, consistiría en permitir alternativamente la ejecución de un escritor por cada lector que finaliza su tarea. Sea cual sea la solución escogida, se debe garantizar que ante un flujo constante de lectores y escritores ninguno de los dos tipos de procesos pueda monopolizar el recurso, impidiendo permanentemente el acceso a los otros y llevando por lo tanto a potenciales situaciones de *deadlock* (bloqueo).

Para el presente monitor, se desea implementar una disciplina que permita un entrelazado eficiente de los procesos escritores y lectores. Las restricciones que se aplicarán al monitor son las siguientes:

- La lectura se podrá realizar de manera concurrente (muchos procesos podrán estar realizando la lectura al mismo tiempo).
- Sin embargo, la escritura se realizará bajo estricta exclusión mutua (cuando un escritor esté leyendo el recurso, no podrá haber ningún otro proceso en él).

Los lectores serán obligados a esperar si hay un escritor dentro del recurso o hay escritores esperando para acceder a él. Si no hay presencia de escritores (ni leyendo ni esperando), el lector podrá tomar el recurso (no importa que haya otros lectores trabajando en él).

Los escritores serán obligados a frenar siempre que haya un proceso dentro del recurso (no importa si lector o escritor), puesto que la escritura se realiza en exclusión mutua.

La finalización de la ejecución de un escritor (y la consecuente liberación del monitor) dará permiso para que todos los lectores que están esperando en la cola puedan acceder y proceder a la lectura del recurso de manera concurrente

El entrelazado de procesos que se pretende conseguir mediante esta disciplina es el siguiente:

1 escritor -> cola de lectores -> 1 escritor -> cola de lectores -> 1 escritor -> ...

La siguiente ilustración detalla la filosofía de funcionamiento que se acaba de describir:

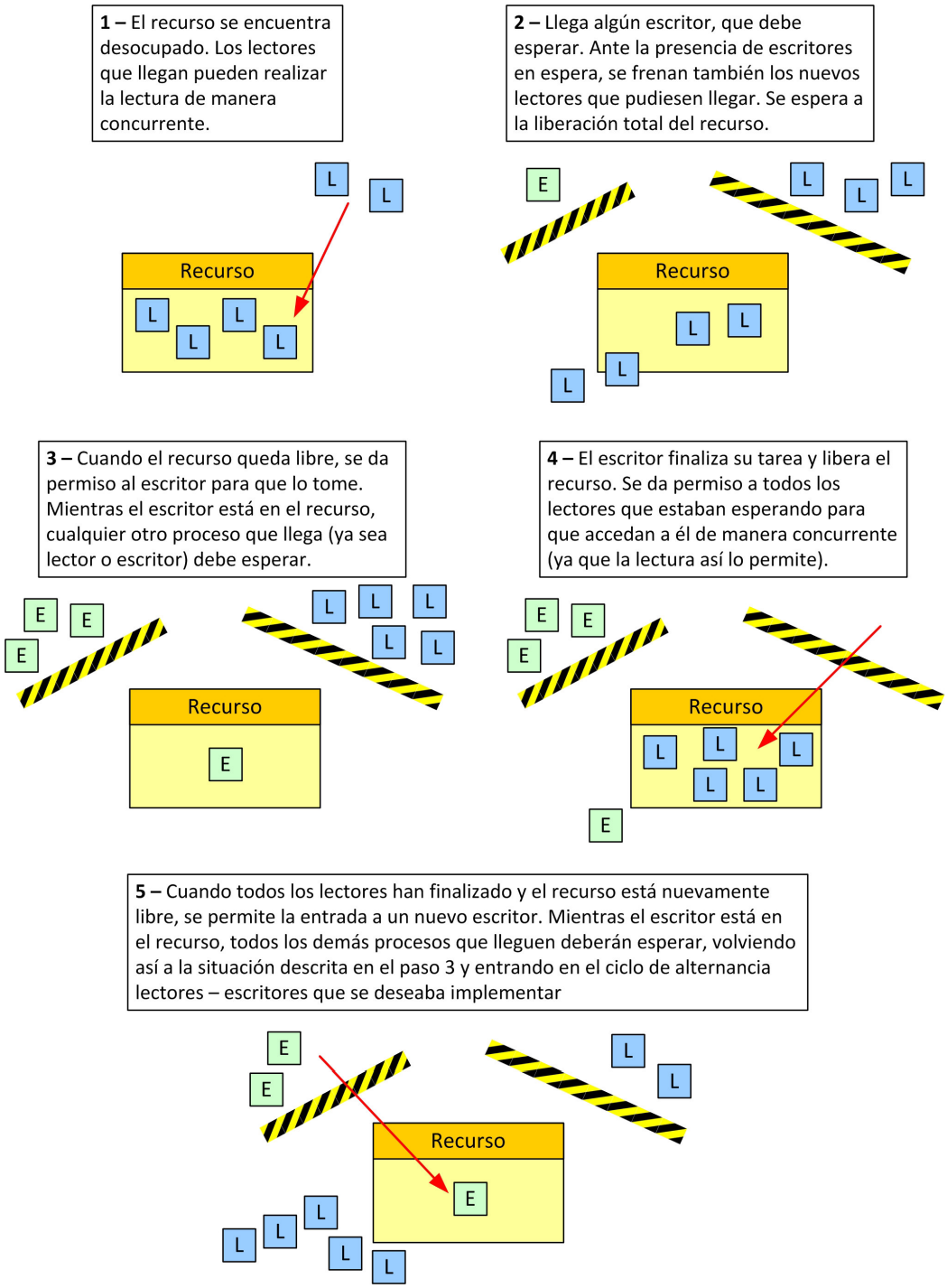

Figura C.1: Disciplina de funcionamiento de un monitor alternativo lectores / escritores

En el lenguaje de programación *Java*, cada objeto dispone de su propia cola de procesos en espera (es decir, cada objeto puede ser usado como un monitor). Es por ello que se utilizará una única cola de espera: la propia del objeto monitor. Todos los métodos de un monitor son declarados *synchronized*, es decir, únicamente un *Thread* podrá estar ejecutando código en el monitor en cualquier momento dado. *Java* proporciona tres métodos básicos para el control de los procesos que son enviados a esperar a dicha cola o desencolados para que continúen con su ejecución.

- `wait()`: envía al proceso que lo invoca a esperar a la cola
- `notify()`: saca uno de los procesos que están en la cola de espera y le concede permiso para continuar su ejecución. La disciplina de la cola de espera es FIFO, por lo que el proceso escogido será aquél que lleve más tiempo esperando.
- `notifyAll()`: desencola todos los procesos de la cola de espera y concede permiso a todos ellos para que continúen su ejecución. Normalmente, cuando se hace uso de este método, algunos de los procesos despertados deberán volver a esperar, mientras que otros ganarán acceso al monitor, dependiendo del estado de las variables de condición. También es habitual el uso de este método para provocar "race conditions" (dejar que los *threads* compitan entre ellos por el acceso al monitor).

El monitor aquí presentado únicamente hace uso de los métodos `wait()` y `notifyAll()`.

C.2. ATRIBUTOS DE CONTROL

Se presentan a continuación los atributos de la clase `SenetBeansMonitor` que ayudarán a establecer las variables de condición, es decir, a determinar cuándo los procesos deben ser frenados o despertados.

```
        private int numeroLectores[];
        private int thisLote;
        private int nextLote;
        private int numeroEscritoresEscribiendo;
        private int numeroTotalEscritores;

        public SenetBeansMonitor() {
                numeroLectores = new int[2];
                thisLote = 0;
                nextLote = 1;
                numeroEscritoresEscribiendo = 0;
                numeroTotalEscritores = 0;
        }
```

Código C.1: Atributos de control de *SenetBeansMonitor*

- `numeroLectores`: *array* de longitud 2 que almacena el número de lectores actualmente leyendo y el número de lectores esperando para leer. Un índice del *array* guarda el número de lectores actualmente activos y el otro el número de lectores en cola. Cuál de los dos índices se está encargando de cada rol es definido por los atributos `thisLote` y `nextLote`.
- `thisLote`: indica qué índice del *array* de contaje de lectores está almacenando en este momento el número de lectores leyendo en el recurso (será forzosamente 0 ó 1).
- `nextLote`: indica qué índice del *array* de contaje de lectores está almacenando en este momento el número de lectores esperando en cola para leer (será forzosamente 0 ó 1, y será el valor complementario al de `thisLote`).
- `numeroEscritoresEscribiendo`: número de escritores que actualmente están escribiendo en el recurso (será forzosamente 0 ó 1, ya que no se permite la presencia de más de un escritor en el recurso).
- `numeroTotalEscritores`: número total de escritores presentes en el monitor (los que están esperando en cola más el que pudiese estar escribiendo en el recurso).

C.3. LECTURA

A continuación se presentan y comentan las implementaciones de los métodos `Leer()` y `salirLectura()`, que deben ser invocados para intentar tomar acceso de lectura al recurso y una vez se ha finalizado el proceso de lectura, respectivamente.

```
public synchronized void Leer() {
        if(numeroTotalEscritores == 0) {
                numeroLectores[thisLote]++;
        }
        else {
                numeroLectores[nextLote]++;
                int myLote = nextLote;
                while(thisLote != myLote) {
                        try {
                                wait();
                        } catch(InterruptedException e) {
                                e.printStackTrace();
                                return;
                        }
                }
        }
}

public synchronized void salirLectura() {
        numeroLectores[thisLote]--;
        if(numeroLectores[thisLote] == 0) {
                notifyAll();
        }
}
```

Código C.2: Implementación de los métodos de gestión de la lectura de *SenetBeansMonitor*

Al invocar el método `Leer()` pueden darse dos casos. Si no hay escritores en el monitor (ni escribiendo ni esperando) se puede acceder al recurso libremente. No se comprueba la presencia de lectores, ya que la concurrencia en lectura está permitida. Al entrar, se marca la presencia del lector incrementando el valor actual de `numeroLectores` (`thisLote` indica el índice del *array* que se debe incrementar).

Por el contrario, si hay presencia de escritores en el monitor, el lector debe esperar a que finalice la ejecución de uno de los escritores. Se marca la presencia del lector en el índice correspondiente del *array* que lleva cuenta de los lectores que están esperando (índice determinado por `nextLote`). Se guarda el número del índice al cual pertenece el lector actual en la variable local `myLote`, y se duerme el proceso mediante llamada a `wait()`. El lector se despertará cuando un escritor salga del monitor y dé permiso a todo un lote de escritores para entrar en él, pero previamente deberá comprobar que efectivamente su número de lote es el que tiene permiso para entrar en el monitor en este turno (si no lo es, deberá volver a esperar).

Al finalizar la lectura e invocar el método `salirLectura()`, el lector se sustrae de la cuenta de lectores que estaban leyendo en el recurso. Si se trataba del último lector del presente lote, se despiertan todos los procesos que están esperando. Si hay escritores esperando, los lectores deberán volver a esperar, porque las variables `thisLote` y `nextLote` no han sido intercambiadas para indicar un cambio en el turno de los lectores. En cambio, alguno de los escritores que se encontraba esperando podrá acceder al recurso.

C.4. ESCRITURA

De manera análoga, se presenta ahora la implementación de las funciones `Escribir()` y `salirEscritura()`, invocadas para obtener acceso de escritura al recurso y para notificar que una operación de lectura ha finalizado y se deben llevar a cabo las acciones oportunas.

```
        public synchronized void Escribir() {
                numeroTotalEscritores++;
                while(numeroLectores[thisLote] + numeroEscritoresEscribiendo != 0) {
                        try {
                                wait();
                        } catch(InterruptedException e) {
                                e.printStackTrace();
                                return;
                        }
                }
                numeroEscritoresEscribiendo = 1;
        }

        public synchronized void salirEscritura() {
                numeroEscritoresEscribiendo = 0;
                numeroTotalEscritores--;
                int temp = thisLote;
                thisLote = nextLote;
                nextLote = temp;
                notifyAll();
        }
```

Código C.3: Implementación de los métodos de gestión de la escritura de *SenetBeansMonitor*

Al invocar el método `Escribir()`, se incrementa la variable que lleva cuenta de los escritores totales presentes en el monitor. El escritor debe esperar si el recurso está ocupado (si ya hay otro *Thread* en el recurso, ya sea un escritor ó varios lectores). Al conseguir entrar en el recurso, el escritor anuncia su presencia poniendo la variable `numeroEscritoresEscribiendo` a 1.

Al finalizar la operación de lectura, el escritor decrementa el número total de escritores en el monitor y marca el recurso como libre de escritura poniendo `numeroEscritoresEscribiendo` a 0. A continuación debe dar permiso a un lote completo de escritores para que puedan acceder de forma concurrente al recurso. Esto lo realiza mediante el intercambio de las variables que indican el número de lectores leyendo y esperando (`thisLote` y `nextLote`), y despertando a todos los procesos. Obsérvese que si el monitor está vacío no sucederá nada, y que si no hay lectores pero sí escritores, uno de ellos tomará el control.

D. MEJORAS SECUNDARIAS INTRODUCIDAS

El objetivo principal de este proyecto es la adaptación de la herramienta de edición y ensamblaje de componentes software CAEAT a un entorno de procesado distribuido, siguiendo una arquitectura orientada a servicios. Sin embargo, el proyecto tiene como objetivo secundario la mejora continua de la herramienta CAEAT. Este anexo describe las mejoras más visibles y de mayor envergadura que se han llevado a cabo paralelamente al desarrollo del grueso del proyecto, entrando ligeramente en los detalles de implementación de dichas mejoras en el lenguaje de programación *Java*.

D.1. ASPECTO DE LOS COMPONENTES DEL TAPIZ

Como se ha detallado en otros capítulos, el aspecto de los componentes del tapiz de CAEAT ha sido remodelado por completo para transmitir rápidamente al usuario cierta información sobre la naturaleza del *bean* o servicio encapsulado en dichos componentes, así como también por razones estéticas. Las principales novedades introducidas se listan a continuación:

- Sustitución de la forma rectangular de los componentes por una forma rectangular con los bordes redondeados, por motivos estéticos.
- Representación en trazo discontinuo del borde de los componentes que encapsulan servicios remotos, para poder identificar rápidamente a estos.
- En el caso de los servicios remotos, inserción de hasta tres iconos alrededor del componente para expresar visualmente diferentes circunstancias y estados que atañen al servicio remoto. El significado y colocación de los iconos se ha expuesto en los capítulos correspondientes.
- En el caso de las patillas de los componentes, se identifican claramente las de salida y las de entrada mediante puntas de flecha orientadas hacia fuera o dentro del componente respectivamente. Se comprobó que las terminaciones redondeadas de las patillas implementadas originalmente resultaban poco intuitivas para la mayoría de los usuarios de la plataforma.

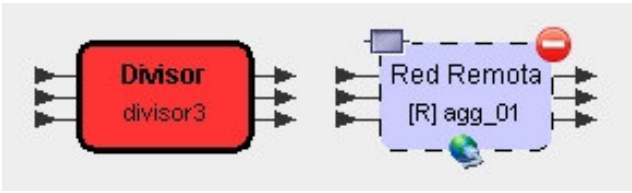

Figura D.1: Ejemplos del nuevo aspecto de los componentes del tapiz

La aplicación del patrón de diseño MVC durante todo el ciclo de desarrollo del presente proyecto resulta especialmente útil a la hora de realizar cambios como el expuesto en este apartado. Puesto que las capas de modelo, vista y controlador están claramente diferenciadas (se trata de clases *Java* distintas), una modificación en el aspecto de los componentes implica únicamente cambios en las clases relacionadas con su vista, dejando inmutable su comportamiento interno.

En el caso de los componentes, la clase encargada de definir su vista se denomina *VistaComponente* y hereda de la clase `JComponent` del paquete *Swing* de Java. Todo componente gráfico de *Java* como `JComponent` contiene un método `paint()` que es llamado por la Máquina Virtual de *Java* cada vez que el elemento debe repintarse en pantalla por alguna razón. Internamente, este método delega en otro llamado `paintComponent(Graphics g)`. Las API's del entorno de programación *Swing* recomiendan sobrescribir este último método si se desea realizar un pintado customizado del elemento.

El método *paintComponent* recibe como parámetro un objeto de la clase `Graphics` que debe usarse para invocar las funciones de pintado deseadas. Mediante esta clase se realiza el dibujado de los componentes, ya que ofrece métodos como los siguientes:

- *drawRect*, *drawRoundRect*, *fillRect*, *fillPolygon* y similares: multitud de métodos que permiten dibujar desde formas básicas hasta figuras y polígonos elaborados. Permiten pintar únicamente los bordes de la figura y / o rellenarla de un color determinado.
- *setFont*, *setStroke* y similares: métodos que permiten elegir el grosor de los trazos a dibujar, su estilo (continuo o discontinuo), la fuente a usar en el caso de dibujar texto, etc.
- *drawImage*: método usado para la representación de los iconos puesto que permite mostrar cualquier imagen en un punto de inserción dado. También se usa para la representación de la imagen del componente, si éste tiene una asociada (ver apartado C.3).

D.2. LOOK AND FEEL DE LA PLATAFORMA

La interfaz gráfica de la plataforma CAEAT ha sido enteramente desarrollada utilizando el paquete *Swing* de *Java*, que ofrece herramientas para la elaboración de interfaces gráficas de usuario de aspecto profesional y realizando abstracción de la plataforma en la cual se ejecutará el software. Una de las características de dicho entorno es que permite el cambio del LAF (*Look and Feel* o apariencia) de la aplicación en tiempo de ejecución, sin necesidad de modificar el código de proceso. Esto es posible porque todas las clases del paquete *Swing* siguen la filosofía del patrón de diseño MVC.

Al inicio del presente proyecto, la plataforma CAEAT estaba dotada del *Look and Feel* por defecto del paquete *Swing*, llamado `MetalLookAndFeel` (véase sub-apartado siguiente). Sin embargo, toda distribución de la Máquina Virtual de *Java* proporciona LAF's alternativos, y además permite el uso sencillo de LAF's desarrollados por terceros. Existen dos razones principales por las cuales se ha decidido experimentar con el cambio de *Look and Feel* de la plataforma CAEAT durante la elaboración de este proyecto:

- **Estética**: el *Look and Feel* de una aplicación define la impresión subjetiva que el usuario se lleva de ella. El LAF debe ser agradable para la mayoría de usuarios a la vez que debe permitir desempeñar todas las interacciones con la interfaz de usuario de una manera cómoda e intuitiva. Un LAF poco agradable puede suponer el fracaso de una plataforma software de calidad, por lo que es conveniente la instalación y prueba de diferentes LAF's en fase de desarrollo.
- **Accesibilidad**: las personas con determinadas deficiencias visuales pueden encontrar dificultoso el manejo de las interfaces de usuario que las plataformas software ofrecen por defecto. Toda plataforma debe ofrecer mecanismos de accesibilidad que permitan establecer LAF's que faciliten la visualización de los contenidos a dichas personas.

D.2.1. Look and Feel's en el lenguaje de programación Java

La instalación de nuevos *Look and Feel's* en una interfaz gráfica desarrollada mediante *Swing* es extremadamente sencilla gracias a la clase `UIManager` y a su método `setLookAndFeel (nombreLAF)`. El nombre del LAF proporcionado a este método corresponde al nombre completo (paquete + clase) de la clase principal que implementa el LAF en cuestión. Dicha clase principal, todas las otras clases que requiera el LAF y todos los recursos gráficos (iconos, imágenes, etc.) deben encontrarse en el entorno de la JVM (en su *classpath*) en el momento del establecimiento del nuevo LAF. Tras establecer un LAF distinto al que se tenía es posible refrescar la interfaz gráfica, de manera que éste se muestre al instante al usuario.

En la plataforma CAEAT se ha implementado un mecanismo de selección de *Look and Feel* mediante el cuadro de selección de opciones que es explicado en el anexo D.4. Al seleccionar un LAF diferente del actual, éste es automáticamente establecido mediante el mecanismo explicado anteriormente y la interfaz gráfica es repintada. Los LAF's proporcionados se pueden dividir en los dos grupos siguientes:

- **LAF's propios de la distribución de *Java***: toda distribución de la Máquina Virtual de *Java* contiene algunos *Look and Feel's* por defecto que es posible instalar en las aplicaciones. La clase `UIManager` ofrece métodos para consultar los LAF's disponibles en cada momento. Por lo general, una distribución reciente de la JVM (versión 1.6 o superior) contiene el LAF por defecto `MetalLookAndFeel`, el LAF `NimbusLookAndFeel` y algunos LAF's adicionales que imitan la apariencia del sistema operativo en el cual se instala la JVM.
- **LAF's elaborados por terceros**: los *Look and Feel's* externos se distribuyen en archivos *jar* y se incluyen en el *classpath* de CAEAT para que la plataforma pueda hacer uso de ellos. Para el presente proyecto se han buscado LAF's de código abierto o con licencias similares a la *Apache License*, de manera que se puedan usar sin restricción en una aplicación que se pretende que sea comercial en el futuro.

En la figura siguiente se pueden observar algunos de los *Look and Feel's* entre los que se permite conmutar la apariencia de la plataforma CAEAT al término del presente proyecto:

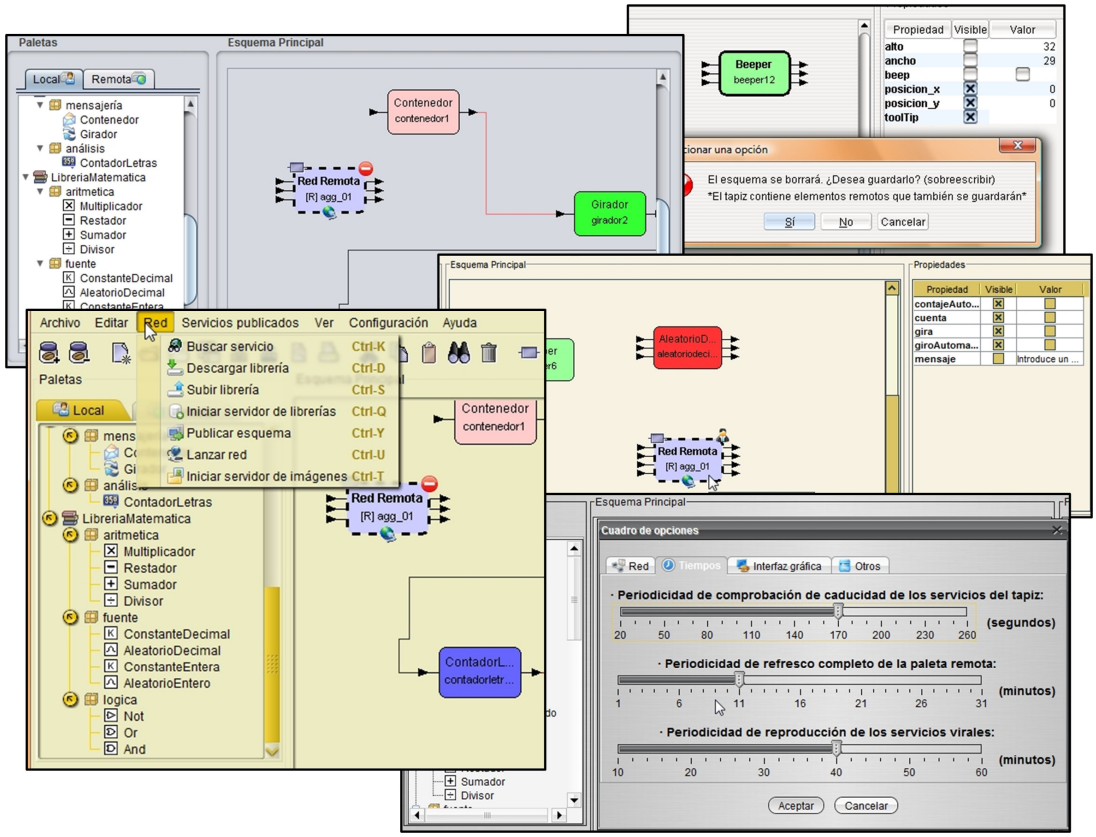

Figura D.2: Algunos *Look and Feel's* alternativos implementados en la plataforma CAEAT

D.2.2. Look and Feel's de alto contraste

Este proyecto ha querido realizar el primer paso hacia la implementación de mecanismos de accesibilidad en la herramienta CAEAT. Dicho paso ha consistido en la implementación de interfaces gráficas de alto contraste. Las interfaces de alto contraste son apropiadas para personas con disfunciones visuales relacionadas con la correcta distinción de la gama cromática, ya que por lo

general están compuestas únicamente de dos colores fácilmente distinguibles (negro sobre fondo blanco, verde sobre fondo negro, etc.).

CAEAT es una plataforma que confía mucho en el coloreado y los parpadeos para informar al usuario de muchas de sus operaciones (distinguir componentes y conexiones, saber si se está en el modo de edición en caliente del tapiz, saber si se está llevando a cabo una búsqueda de servicios, etc.). Es por ello que se han implementado tres juegos de colores de alto contraste:

- Elementos negros sobre fondo blanco
- Elementos blancos sobre fondo negro
- Elementos verde fósforo sobre fondo negro

Las interfaces de alto contraste, en realidad, no se han implementado como nuevos *Look and Feel's*. La programación de un LAF propio es una operación compleja que implicaría la creación de todo un paquete de clases *Java* nuevo, así como de todos los recursos que el LAF necesitase utilizar. En lugar de eso, se ha aprovechado el hecho de que los LAF's del lenguaje de programación *Java* pueden ser asociados a un *Theme*.

En desarrollo software, un *Theme* es un paquete predefinido que contiene los detalles de la apariencia gráfica que se usarán para personalizar un *Look and Feel*. En el lenguaje de programación *Java*, un *Theme* es una clase que implementa unos métodos bien definidos que proporcionan detalles básicos acerca de la apariencia tales como los colores de fondo y de primer plano de las ventanas, el tipo y tamaño de la fuente de los textos, los iconos a mostrar en los diálogos básicos, los colores de selección de los menús, etc. Estas informaciones son consultadas por el mecanismo de repintado de la JVM cada vez que es necesario refrescar la interfaz gráfica.

EL LAF por defecto de *Swing*, `MetalLookAndFeel`, está asociado por defecto a una clase llamada `DefaultMetalTheme` que se encarga de proporcionar esta información. Para la definición de los juegos de colores de alto contraste, se han diseñado clases que heredan de ésta última y que devuelven los colores apropiados para cada interfaz de alto contraste en lugar de los colores por defecto de *Swing*.

Los juegos de colores de alto contraste son presentados al usuario en el cuadro de selección de opciones como si fuesen LAF's distintos, pese a que en realidad al cambiar a ellos se instala en la plataforma el LAF por defecto `MetalLookAndFeel` asignándole el *Theme* customizado que corresponda a la selección del usuario. A continuación se muestran capturas que muestran parcialmente los tres juegos de color de alto contraste implementados:

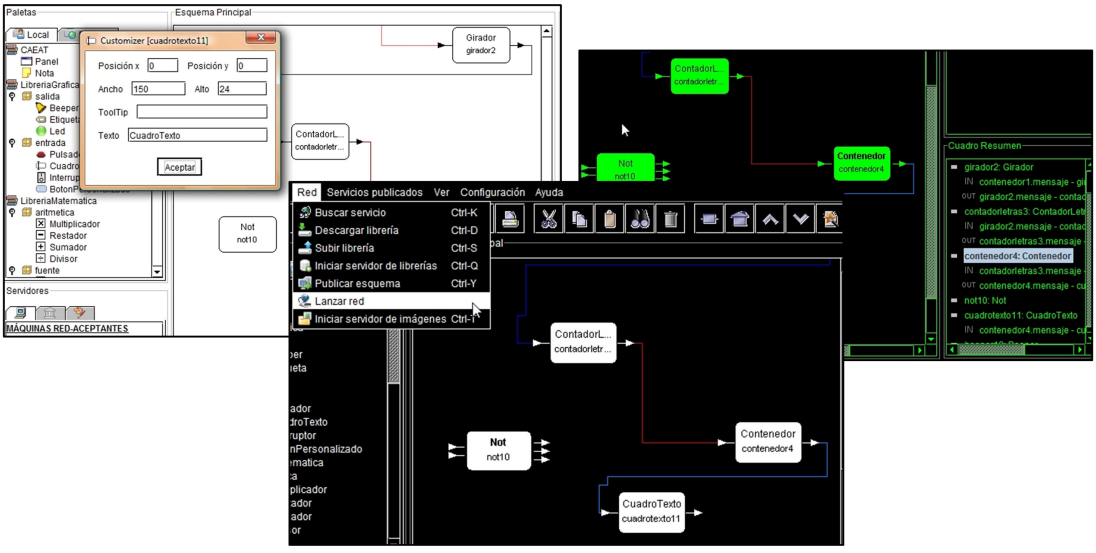

Figura D.3: Juegos de color de alto contraste implementados en la plataforma CAEAT

En el marco de la plataforma CAEAT es posible realizar más acciones que ayuden a mejorar la accesibilidad a la herramienta, como la elección del tamaño de fuente de los textos de las interfaces y los menús. Estas modificaciones restan como línea de trabajo abierta. Algunas recomendaciones y normativas, tanto a nivel nacional como internacional, acerca de los mecanismos de accesibilidad en las plataformas software pueden consultarse en las referencias [**15**] y [**16**].

D.3. ASOCIACIÓN DE IMÁGENES A LOS COMPONENTES DEL TAPIZ

Por lo general, los componentes del tapiz muestran dentro de sí el nombre del *bean* o servicio al cual pertenecen ("Sumador", "Socket", etc.), así como el nombre indexado de la instancia de ese componente en el tapiz de CAEAT ("sumador1", "socket2", etc.). Uno de los objetivos secundarios de este proyecto consiste en mejorar el proceso de ensamblaje de una agregación permitiendo que éste sea mucho más rápido, visual e intuitivo.

Para conseguir tal fin se ha implementado un mecanismo que sustituye la información textual de los componentes del tapiz por imágenes escogidas por el usuario que representan al *bean* o servicio que está ejecutándose en dicho componente. Tal y como se explicó en el anexo D.1, la representación de imágenes en un objeto *Java* que hereda de la clase `JComponent` es posible mediante la apropiada reescritura del método `paintComponent(Graphics g)`; de la misma manera en que se representan los iconos informativos alrededor del componente.

Dado que los componentes del tapiz tienen un tamaño de 80 x 50 píxeles, se admiten como válidas imágenes de tamaños 32 x 32 y 48 x 48, dimensiones estándares para las cuales resulta sencillo encontrar paquetes de imágenes de distribución gratuita. Dichas imágenes se almacenan en la estructura de ficheros interna de CAEAT, en un directorio específico junto con las imágenes propias de la plataforma (iconos y demás).

Para facilitar su inserción en los componentes del tapiz, se ha habilitado un acceso directo en el menú contextual de los componentes. Al pulsar sobre él se analizan todos los archivos contenidos en la carpeta destinada a las imágenes y se descartan los no válidos (únicamente se consideran imágenes de los tamaños anteriormente especificados y de tipo *png*, *gif* y *jpg*). Las imágenes válidas son mostradas en una interfaz de selección.

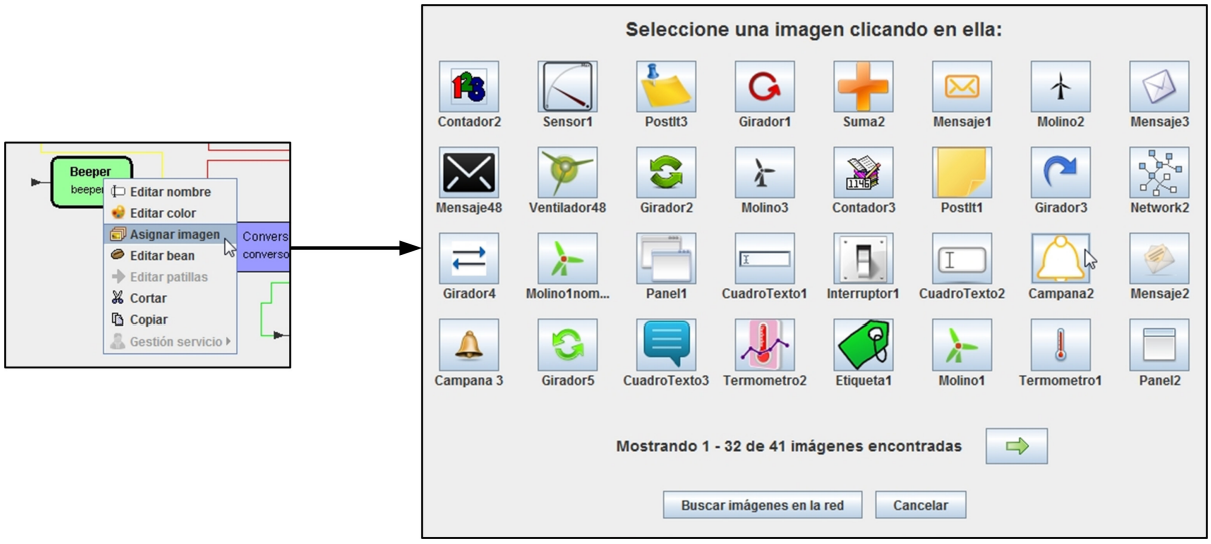

Figura D.4: Acceso a la interfaz de selección de imágenes de tapiz a través del menú contextual

En la interfaz de selección, las imágenes se muestran insertadas en botones de manera que la pulsación sobre uno de ellos provoca la asociación inmediata de la imagen con el componente bajo edición. La situación más habitual será la de no disponer en el entorno local de imágenes que concuerden con lo deseado por el usuario. En este caso, es posible realizar una búsqueda de imágenes por la red usando los servidores de imágenes descritos en el capítulo 8.2.

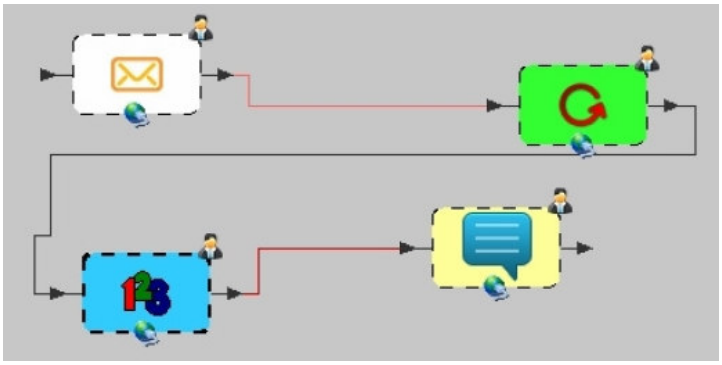

Figura D.5: Agregación de componentes con imágenes asociadas a los mismos

El guardado en un archivo *aec* del esquema del tapiz implica también un guardado de las imágenes que pudiesen estar asociadas a los componentes de dicho esquema. Para no complicar en exceso el esquema de guardado, las imágenes son convertidas a un *buffer* de *bytes* y guardadas en el propio archivo *aec* junto con los *beans* a los cuales están asociados. La conversión de las imágenes a un *buffer* de *bytes* requiere de un tratamiento particularizado de la imagen atendiendo a su tipo. Por lo general, cada tipo de imagen utiliza un modelo de coloreado diferente (por ejemplo, los archivos *png* admiten un parámetro de transparencia en sus píxeles, mientras que los *jpg* no). El lenguaje de programación *Java* ofrece clases y herramientas para afrontar estas particularidades.

D.4. CUADRO DE OPCIONES

Durante toda la fase de desarrollo del presente proyecto se ha requerido la toma de decisiones acerca del valor de algunas magnitudes, o de selección de algunas rutas de almacenamiento de recursos. En fase de desarrollo, se ha realizado *hardcoding* con estos valores (se han introducido directamente en el código fuente), pero una vez finalizado el producto se desea proporcionar al usuario la libertad de elegir estos parámetros a su voluntad.

Para ello, se ha implementado un cuadro de opciones muy acorde al de otras plataformas de edición gráfica similares a CAEAT. Dicho cuadro de opciones es accesible mediante una entrada de menú de CAEAT, y está organizado en pestañas según se detalla a continuación. Las opciones seleccionadas durante la ejecución de CAEAT son permanentes, puesto que los nuevos valores de los parámetros son escritos en un fichero de propiedades que es cargado al inicio de cada ejecución de la plataforma.

D.4.1. Opciones de red

La figura siguiente muestra la pestaña de selección de opciones de red:

Figura D.6: Pestaña de selección de opciones relativas a operaciones de red

Permite escoger tres parámetros. El primero es el puerto en el cual CAEAT asume que hay un servidor HTTP aceptando conexiones y sirviendo los recursos adecuados. Para una correcta publicación de los servicios, debe coincidir con el puerto escogido en el servidor HTTP que se haya desplegado en la máquina.

El segundo parámetro corresponde a la ruta hasta la carpeta pública en la cual se copiarán los recursos de los servicios publicados desde la máquina local (archivos *class*, iconos e imágenes). Lógicamente, esta carpeta debe ser uno de los directorios "HOME" del servidor HTTP que se sirvan a los clientes. Por defecto, este directorio apunta a una carpeta llamada "CAEAT-Public" ubicada dentro de la estructura de ficheros de CAEAT, pero se debe dar a un usuario experimentado la oportunidad de elegir cualquier carpeta de su sistema de ficheros.

El último parámetro corresponde al puerto en el cual *CAEAT Service Manager* registra su interfaz RMI para ponerla a disposición de futuras instancias de CAEAT y permitir así que éstas lo gobiernen. En la práctica, basta con la elección de un puerto disponible en la máquina que se sepa con certeza que no será usado por ningún otro servicio. Si CSM ya ha sido iniciado en la máquina local, no se permite la modificación de este parámetro, ya que al no encontrarse la interfaz de CSM en el nuevo puerto se procederá al lanzamiento de un nuevo CSM, y la ejecución de dos gestores de servicios en la misma máquina es una situación anómala y no deseada.

D.4.2. Opciones de temporización

La figura siguiente muestra la pestaña de selección de diferentes tiempos de espera:

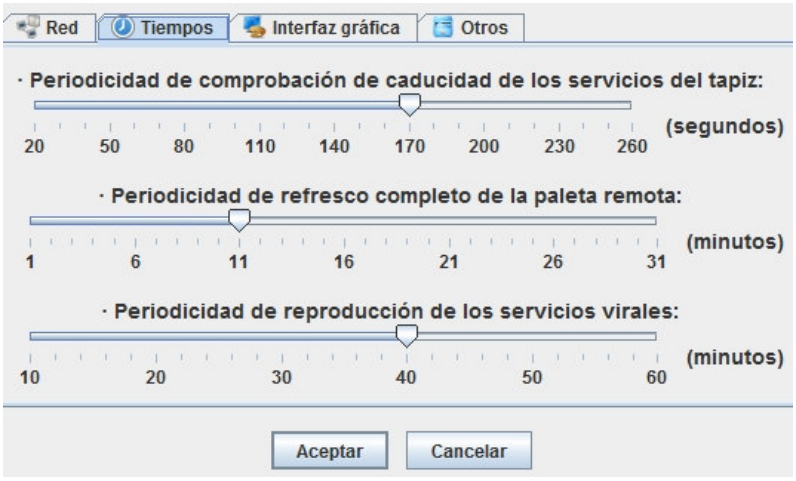

Figura D.7: Pestaña de selección de opciones relativas a los diferentes tiempos de espera

Esta pestaña permite la elección mediante *slidebars* de tres tiempos de espera. El primero especifica, en segundos, cuánto tiempo debe esperar el *Thread* que realiza comprobaciones periódicas de la supervivencia de los servicios actualmente insertados en el tapiz entre comprobaciones sucesivas. El segundo corresponde a la periodicidad de refresco de la paleta remota (tanto la de servicios como la de servidores y repositorios). De manera similar al caso anterior, el *Thread* encargado de realizar este cometido esperará inactivo el número de minutos especificado. Por último, se permite escoger la periodicidad con la que el *Thread* encargado de expandir los servicios virales se activa y trata de realizar su expansión mediante los procedimientos que se explicaron en el capítulo 6.5.

D.4.3. Opciones de interfaz gráfica

La figura siguiente muestra la pestaña de selección de opciones relativas a la interfaz de usuario:

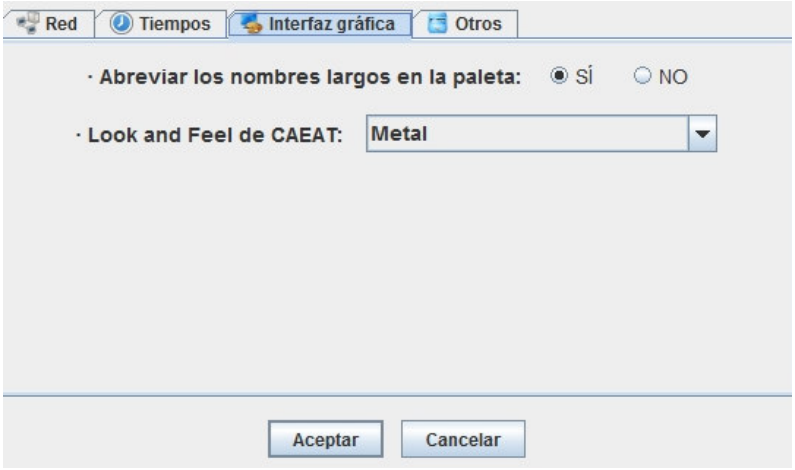

Figura D.8: Pestaña de selección de opciones relativas a la interfaz de usuario

Permite elegir dos parámetros de visualización. El primero hace referencia a la longitud de los nombres de *beans* y servicios en la paleta. Si los nombres superan el ancho establecido de la paleta, pueden ser truncados para ajustarse a ésta o dejarse tal y como son y añadir una barra de desplazamiento horizontal a la paleta para poder leerlos al completo.

También se proporciona en esta pestaña el menú desplegable que permite cambiar el *Look and Feel* de la plataforma. Los LAF's ofrecidos por el desplegable están formados por aquéllos incluidos en la distribución de la Máquina Virtual de *Java* actualmente en uso, los LAF's de código abierto generados por terceros e incluidos en el proyecto, y los tres juegos de colores de alto contraste desarrollados. Siguiendo los mecanismos explicados en el anexo D.2, el nuevo LAF es instalado al instante y la interfaz gráfica es inmediatamente repintada en el momento en que el usuario cambia el contenido del menú desplegable, consiguiéndose de esta manera un acabado muy visual para el usuario y de aspecto profesional.

D.4.4. Otras opciones

La figura siguiente muestra una pestaña que se ha habilitado para la inclusión de otras opciones:

Figura D.9: Otras opciones disponibles en el diálogo de opciones

Por el momento, en esta pestaña se ha habilitado únicamente la opción de selección del directorio en el cual se almacenan los archivos *jar* de las librerías que se van añadiendo a CAEAT. Por defecto, este directorio es interno a la estructura de carpetas de CAEAT, pero se debe dar al usuario la libertad de elección de cualquier directorio de su sistema de ficheros.

Todas las pestañas presentadas son susceptibles de ser ampliadas en el futuro con más opciones, a medida que el crecimiento de la plataforma así lo requiera.

D.5. GESTIÓN DE USUARIOS

En anteriores capítulos se ha expuesto el concepto de "propietario" de un servicio, lo cual lleva a la necesidad de realizar una distinción entre usuarios de CAEAT. El nombre de usuario de CAEAT consiste en una variable global de tipo *String* que se establece al inicio de la ejecución y permanece durante toda la sesión de trabajo. Esta variable resta asociada a los servicios desplegados por el usuario, tanto en los objetos publicados en los servidores LUS como en su representación en el documento XML descriptor de las agregaciones.

Durante la elaboración del presente proyecto se ha implementado un sencillo mecanismo de control y gestión de usuarios consistente en una pantalla de *login* al inicio de la ejecución de CAEAT que pide al usuario un nombre y una contraseña. Todos los servicios creados y desplegados durante la sesión de trabajo serán posesión del usuario que ha realizado *login*, y será él quien posea los derechos de modificación de su ciclo de vida (finalización, des-publicación, reinicio, etc.).

También es posible el cambio de usuario "en caliente", sin necesidad de detener la ejecución de la plataforma, mediante la realización de un *logoff* a través de una entrada de menú y previa limpieza de los servicios y componentes alojados en el tapiz de la herramienta:

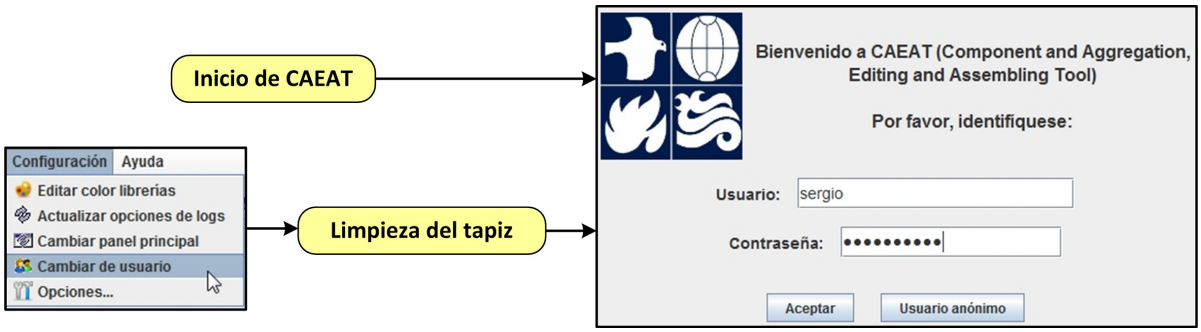

Figura D.10: Interfaz de introducción de credenciales de usuario

Se ha dejado como línea de trabajo abierta la implementación de un mecanismo de control de usuarios distribuido (ver capítulo 9.2.1.6). En la actualidad, las credenciales de cada usuario (nombre y contraseña) se almacenan de manera local en el fichero de propiedades de la plataforma, y los nuevos usuarios deben darse de alta mediante edición manual de dicho fichero. En un entorno distribuido como el que constituyen los servicios de *SeNetComponents* y la herramienta de edición CAEAT, este tipo de información deberá ser almacenada y consultada en una o varias entidades distribuidas por la red, que se encargarán de velar por la consistencia de los datos de usuario a lo largo de toda la federación.

De manera similar, se deberá implementar también un juego de privilegios, de manera que cada usuario tendrá acceso a la realización de diferentes acciones en lo que respecta a la obtención, edición y publicación de servicios. Existirá también un perfil de usuario anónimo con menores privilegios que el resto de sus compañeros. Consúltese el capítulo de líneas abiertas de trabajo para más detalles.

E. PATRONES DE DISEÑO SOFTWARE INTRODUCIDOS

En el capítulo 2.3, dedicado a la filosofía de diseño software que se sigue en la plataforma CAEAT, se expusieron los patrones de diseño empleados durante la elaboración de la versión de la plataforma precedente al inicio del presente proyecto. Se detallaron los patrones de diseño MVC (*Model – View – Controller*) y el patrón *Observer – Observable*.

Los patrones de diseño son soluciones clásicas a problemas y casuísticas que se suelen dar con frecuencia en el proceso de desarrollo software mediante la programación orientada a objetos. Por ello, este proyecto tenía como objetivo secundario la aplicación de nuevos patrones de diseño para simplificar las futuras ampliaciones de la plataforma.

Además de respetar los patrones de diseños introducidos en la fase inicial de desarrollo de la plataforma, este proyecto ha introducido los patrones que se presentan a continuación. Para cada uno de ellos se describe a alto nivel su utilidad y sus casos de uso, para a continuación ejemplificar su utilización en la plataforma CAEAT con extractos de código.

E.1. FACTORY

El patrón de diseño *Factory* es de tipo creacional, es decir, sirve para la creación de nuevos objetos sin la necesidad de invocar explícitamente a su constructor mediante el uso de *new*. Es útil para obtener instancias de una clase conociendo únicamente la interfaz que implementa dicha clase, y sin necesidad de especificar el tipo de clase concreto que se creará y retornará (que puede depender de distintas circunstancias).

Este patrón de diseño introduce la figura de la factoría, una clase a la cual un cliente puede solicitar la creación de objetos haciendo abstracción del proceso de creación interno que se ha implementado. La clase retornada implementará una interfaz conocida por el cliente, por lo que podrá ser utilizada por éste, pero será de un tipo u otro atendiendo al "proceso de fabricación" (los detalles de implementación encapsulados por la factoría y escondidos al cliente).

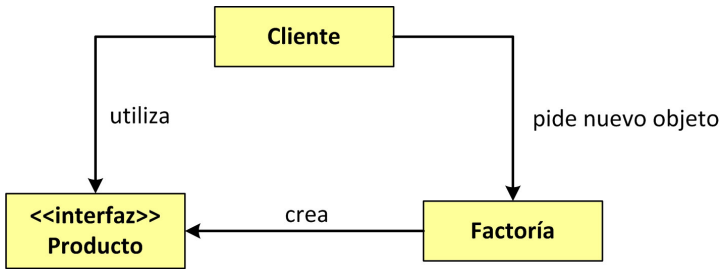

Figura E.1: Representación esquemática del patrón de software *Factory*

E.1.1. Aplicación a CAEAT

Las librerías de la plataforma CAEAT hacen uso del patrón *Factory* para la creación de nuevas instancias de los *proxies* de los servicios. La figura que actúa como cliente es la clase *Publicador*, que tiene como responsabilidad la obtención de una nueva instancia de la clase *Proxy* y su publicación en los servidores de *Lookup*. Por su parte, la figura que actúa como factoría en este caso es la propia clase *Proxy*, que devuelve una nueva instancia de sí misma que puede ser customizada atendiendo a diferentes circunstancias.

En el siguiente extracto de código se muestra la aplicación del patrón desde el punto de vista de la clase *Publicador* (el cliente que solicita el objeto a la factoría). En un momento dado durante la publicación de un servicio, una instancia de la clase *Servidor* es exportada para la recepción de

llamadas remotas, dando como resultado una interfaz remota (`inter`) que implementa la interfaz del servicio (`ContadorLetrasInterface`).

Esta interfaz remota es utilizada para la creación del objeto *Proxy* que se registrará en los servidores de *Lookup*. En lugar de invocar explícitamente el constructor de la clase `ContadorLetrasProxy`, se utiliza su método estático `crear` pasándole como parámetro la interfaz remota recién creada. La entidad que ha solicitado la creación recibe como resultado otro objeto que implementa la interfaz del servicio, que actuará como *proxy* a registrar en los servidores de *Lookup*.

```
ContadorLetrasInterface inter;
ContadorLetrasInterface proxy;

(…)

inter = (ContadorLetrasInterface)bje.export((ContadorLetrasServer)beanServer);

(…)

proxy = ContadorLetrasProxy.crear(inter);
```

Código E.1: Creación de una nueva instancia de la clase *Proxy* desde la clase *Publicador*

Nótese sin embargo que la entidad solicitante ha hecho total abstracción del proceso de creación del objeto, que ha recaído sobre la clase `ContadorLetrasProxy`. El objeto retornado es utilizable porque implementa la interfaz del servicio, pero puede ser de distintos tipos, o puede haber sido configurado bajo ciertos parámetros de manera transparente a la entidad que ha solicitado su creación.

A continuación se presenta el código de la otra figura implicada en la creación del objeto: la factoría. Obsérvese que el método estático `crear` actúa de factoría y es el encargado de invocar explícitamente al constructor de la clase que se desea instanciar.

```
        public ContadorLetrasProxy(ContadorLetrasInterface inter) {
                this.inter = inter;
                this.name = "Contador de letras en un mensaje";
                this.version = "1.0";
                this.manufacturer = "Sener";
                this.vendor = "Sener";
        }

        public static ContadorLetrasProxy crear(ContadorLetrasInterface inter) {
                return new ContadorLetrasProxy(inter);
        }
```

Código E.2: Implementación sencilla de un método factoría para la creación de objetos *Proxy*

Las librerías implementadas durante la elaboración del presente proyecto implementan el patrón *Factory* para la creación de objetos *Proxy*, pero como se puede observar en el extracto de código anterior, el método que lleva a cabo la creación del objeto no realiza ningún tipo de criba ni procesado, y se limita a invocar el constructor del objeto. Sin embargo, en una futura ampliación de las librerías que incluyera la posibilidad de trabajar con distintos tipos de *proxies*, el método `crear` sería el encargado de discernir cuál de ellos se debe retornar atendiendo a los parámetros adecuados.

Una de las líneas de trabajo abiertas consiste en la implementación de restricciones de integridad, confidencialidad, autenticación, etc. en las llamadas remotas que se realizan sobre los *proxies*. El método factoría podría imponer al objeto *proxy* retornado unas u otras restricciones atendiendo a diferentes circunstancias (criticidad del servicio, nivel de seguridad de la máquina que lo está desplegando, etc.), siendo estas restricciones transparentes para la entidad que solicita la creación del *proxy* y que lleva a cabo su publicación.

E.2. SINGLETON

El patrón de *Singleton*, del mismo modo que el *Factory*, es un patrón relacionado con la creación y el instanciado de nuevos objetos. Este patrón restringe el número de instancias de una clase que pueden existir al mismo tiempo en una aplicación a un único objeto. Resulta útil cuando una aplicación, ya sea debido a restricciones de diseño o de rendimiento, consigue resultados más satisfactorios si se utiliza únicamente un objeto de la clase en cuestión. El patrón de *Singleton* puede ser fácilmente extendido para restringir el número de instancias existentes de una misma clase a cualquier número dado.

Del mismo modo que el patrón *Factory*, la creación de un objeto mediante *Singleton* evita la invocación directa al constructor de la clase. Para su implementación, la clase a instanciar debe proporcionar un método estático accesible sin la necesidad de la existencia de una instancia de dicha clase. Al invocar la entidad solicitante del objeto dicho método, la clase a instanciar comprueba si ya existe un objeto de dicha clase. En caso de no existir, es creado en ese momento, en caso de existir retorna el objeto ya existente sin crear uno nuevo.

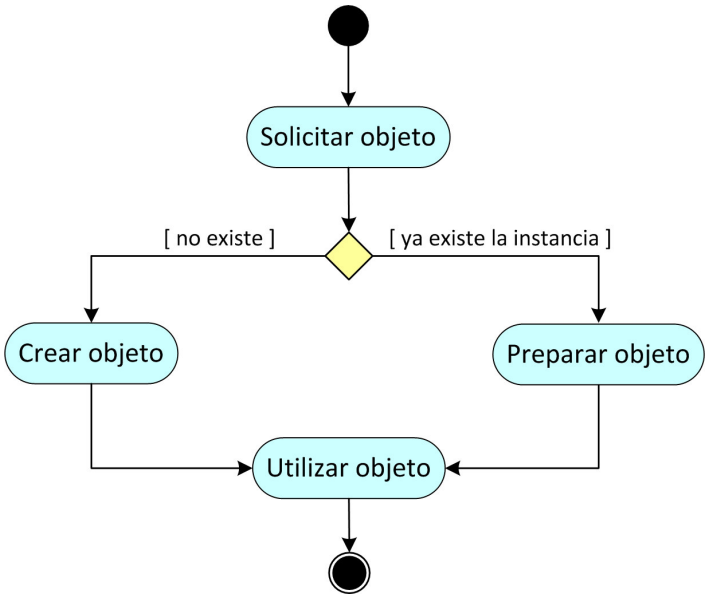

Figura E.2: Representación esquemática del patrón de software *Singleton*

E.2.1. Aplicación a CAEAT

Los detalles de implementación de la búsqueda de servicios en CAEAT (ya sean servicios utilizables en el tapiz, repositorios de librerías, servidores de imágenes, etc.) son encapsulados por unas clases creadas únicamente para tal fin (`BuscadorServicios`, `BuscadorLibrerias`, etc.). Cuando se solicita una búsqueda, estas clases realizan todas las tareas de búsqueda de los servidores de *Lookup*, de comunicación con ellos y de obtención de los objetos públicos de los servicios registrados en ellos.

El objeto buscador, pues, almacena una gran cantidad de objetos como resultado de la búsqueda de servicios por la red y el manejo de los objetos de *Jini / Apache River* que sirven para tal fin. Las partes de la plataforma que deben mostrar los resultados de las búsquedas (por ejemplo, los controladores de las distintas paletas de servicios de CAEAT) no realizan ninguna búsqueda, simplemente consultan los resultados obtenidos por el buscador correspondiente.

En un escenario como el presentado, la utilización del patrón de *Singleton* para la creación de los objetos buscadores resulta útil en términos de eficiencia y escalabilidad del código por las siguientes razones:

- En la plataforma CAEAT únicamente se muestran al usuario los resultados de una búsqueda al mismo tiempo, y una búsqueda no es iniciada hasta que finaliza la anterior, por lo que no tiene sentido mantener diversas instancias del objeto buscador.
- En una federación con gran cantidad de servicios publicados, el objeto buscador de servicios puede llegar a crecer mucho en términos de memoria, ya que almacena los objetos *proxy* de todos los servicios encontrados. No resultaría eficiente sobrecargar la Máquina Virtual de *Java* con distintas instancias del objeto buscador si por el razonamiento anterior únicamente se utilizará uno al mismo tiempo.
- Los resultados de las búsquedas deben poder ser consultados rápidamente por diferentes clases de la plataforma, que a su vez pueden solicitar nuevas búsquedas. Resulta mucho más eficiente ofrecer un mecanismo centralizado para obtener el único objeto existente y poder consultar sus atributos y llamar a sus métodos que instanciar un nuevo objeto por cada módulo del software que necesita hacer uso de él.

La implementación del patrón *Singleton* en la clase `BuscadorServicios` de la plataforma CAEAT es la siguiente:

```
private static BuscadorServicios buscador = null;

(…)

public static BuscadorServicios getBuscador(ControladorRed ctrlRed, CAEAT caeat) {
        //Factoría de buscadores que implementa el patrón de diseño de Singleton
        if(buscador == null)
                buscador = new BuscadorServicios(ctrlRed, caeat);
        else
                buscador.limpiarVariables();
        return buscador;
}
```

Código E.3: Aplicación del método de creación de objetos *Singleton* a la clase *BuscadorServicios*

La clase `BuscadorServicios` almacena una referencia estática a una instancia de sí misma, que inicialmente se establece a `null`. Cuando la primera de las entidades que desea obtener una referencia al buscador realiza la llamada al método estático *getBuscador*, se comprueba si ya existe un objeto de este tipo creado. Si no lo hay, se instancia adecuadamente uno nuevo y se retorna. Si ya existe (porque ya se había producido una llamada a *getBuscador* previamente) se llama a un método que inicializa algunas variables de proceso si es necesaria dicha inicialización, para a continuación devolver el objeto.

E.3. ADAPTER

El patrón de diseño *Adapter* sirve para "traducir" los métodos ofrecidos por una interfaz implementada por una clase a los métodos comprensibles por otra clase. Este patrón de diseño también es denominado en ocasiones *Wrapper* por su función de "envoltura" de la entidad adaptada alrededor de una entidad adaptadora comprensible por el cliente.

La entidad adaptadora permite la colaboración y la comunicación entre conjuntos de clases que normalmente serían incompatibles debido al desconocimiento mutuo de sus interfaces. Esta entidad, conocida por el cliente, sencillamente traduce las llamadas sobre los métodos de su propia interfaz a llamadas sobre los métodos de la interfaz de la entidad adaptada. También puede realizar funciones de adaptación de los datos retornados por las llamadas. Por ejemplo, si la entidad adaptada almacena una serie de valores de tipo *boolean* (*flags*) como un único entero binario, pero los clientes necesitan un valor *true* / *false*, la entidad adaptadora sería la responsable de la obtención y el retorno del valor *boolean* concreto a partir del entero almacenado por la entidad adaptada.

A continuación se presenta un diagrama del patrón de diseño *Adapter* que ejemplifica lo expuesto:

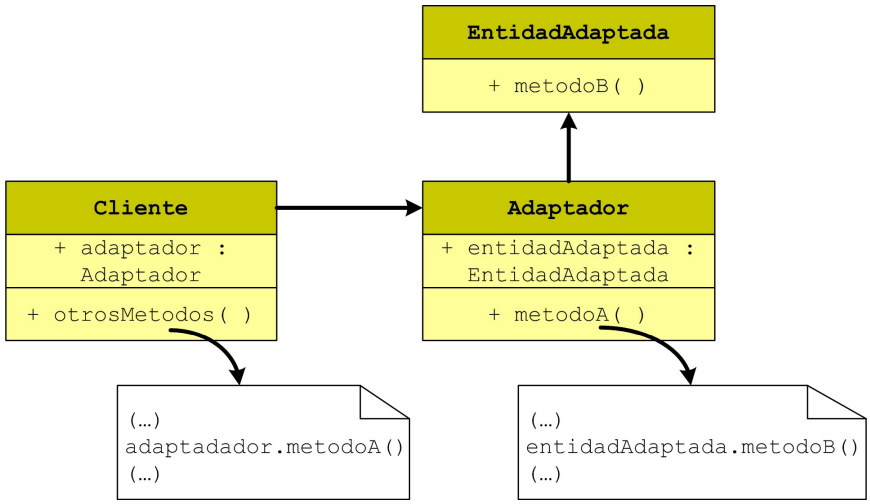

Figura E.3: Representación esquemática del patrón de software *Adapter*

Nótese que la entidad cliente contiene una instancia de la clase adaptadora. El cliente desea utilizar el `metodoB` de la entidad adaptada, pero desconoce su interfaz y /o los tipos de datos que acepta y retorna. En su lugar, invoca el `metodoA` del adaptador, y será esta entidad la que delegue la llamada en el `metodoB` después de realizar el procesado y la adaptación de datos necesarios (si se requiere dicho paso).

E.3.1. Aplicación a CAEAT

El hecho de que al patrón de diseño *Adapter* se le acostumbre a denominar *Wrapper* lleva a pensar que éste es el patrón que se utiliza en la clase *Wrapper* de las librerías de componentes remotos, cuya función es precisamente la de adaptar la semántica de los objetos *Proxy* de *Jini / Apache River* a la semántica de un componente comprensible por CAEAT (un *bean* que sigue la filosofía impuesta por el estándar *JavaBeans*).

En el entorno de CAEAT, la entidad cliente sería la propia plataforma CAEAT, que contiene una gran cantidad de entidades adaptadoras. Estas entidades adaptadoras son los *wrappers* de los servicios obtenidos y actualmente en uso. Dado que las llamadas directas sobre el objeto *Proxy* no son posibles porque éste no sigue la semántica de *JavaBeans*, éstas se realizan sobre un objeto *wrapper*, también obtenido desde los servidores de *Lookup*, que sí la siguen.

Los *wrappers* encapsulan a los *proxies*, por lo que "traducen" las llamadas realizadas sobre ellos a las llamadas a los métodos equivalentes en los *proxies*. Este esquema es idéntico al presentado a alto nivel en la ilustración del apartado anterior. Si se debe realizar alguna adaptación también en el formato de los datos, se realiza en la clase *wrapper* previamente y/o posteriormente a la delegación de la llamada en el objeto *proxy*.

La siguiente figura muestra el esquema del patrón de diseño *Adapter* aplicado a la estructura de clases de CAEAT. Nótese que en este caso la entidad adaptadora o *wrapper* está delegando la invocación de `getValor` en un método llamado exactamente igual, pero realmente la llamada podría delegar en un método completamente diferente, e incluso realizar llamadas a dos métodos distintos y combinar y procesar sus resultados antes de devolverlos a CAEAT.

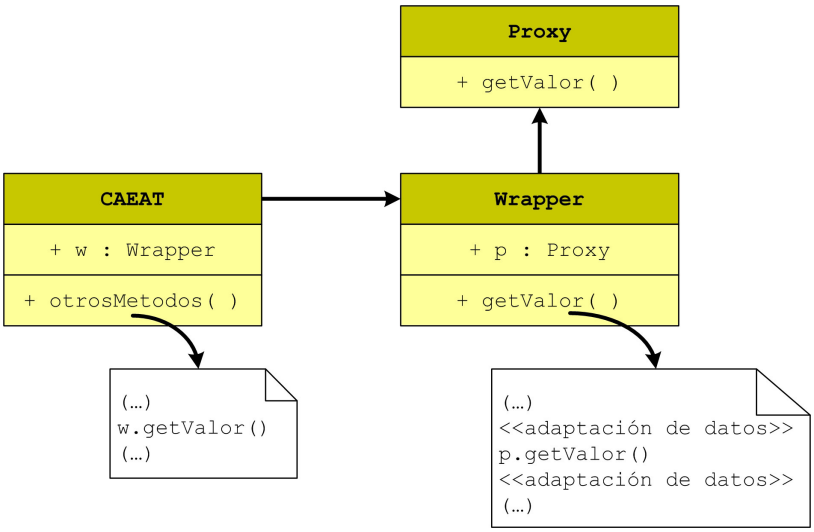

Figura E.4: Aplicación del patrón de diseño *Adapter* al entorno CAEAT

Los *wrappers* de las librerías desarrolladas durante la elaboración del presente proyecto no realizan en la práctica ninguna adaptación de datos, y se limitan a redireccionar la llamada al método que se realiza sobre ellos hacia el método correspondiente del *proxy* del servicio. Sin embargo, debe notarse que la aplicación del patrón *Adapter* deja la puerta abierta a la utilización de los servicios desarrollador por plataformas totalmente distintas a CAEAT. Simplemente se deberían implementar los *wrappers* o adaptadores adecuados capaces de traducir la semántica dependiente de las nuevas plataformas a las llamadas correspondientes sobre los *proxies* de los servicios.

F. PLATAFORMA DE PRUEBAS

Durante el desarrollo de una aplicación software que lleva a cabo un procesado distribuido como es el caso de CAEAT / *SeNetComponents*, se debe hacer especial hincapié en la realización de un testeo en un escenario lo más parecido posible al entorno de producción definitivo en el cual se desplegará la plataforma. Esto es necesario por el hecho de que una aplicación distribuida es generalmente desarrollada en una única máquina de trabajo y bajo un único entorno y sistema operativo, y algunas vicisitudes que pueden aparecer al desplegar la aplicación en diferentes entidades de naturaleza muy diversa podrían quedar ocultas.

En el caso de CAEAT / *SeNetComponents*, además, se tiene especial interés en averiguar los requisitos mínimos de funcionamiento de las herramientas desarrolladas en términos de configuración de red. La aplicación hace uso de una gran cantidad de funcionalidades que podrían ser bloqueadas por los *firewalls* instalados en los *routers* o los diferentes sistemas operativos. También se pretende experimentar con la resolución de nombres de los *hosts* de la red (DNS) y con la visibilidad de los paquetes y los mensajes entre las diferentes entidades, para averiguar cómo una mala configuración de estas características puede lastrar el buen funcionamiento del software desarrollado.

Este apartado realiza una descripción de la plataforma de pruebas empleada para el testeo de la herramienta CAEAT y los servicios y programas que la rodean. En primer término se presenta la arquitectura que se pretende implementar, el material y los recursos (tanto hardware como software) utilizados para desplegarla y el resultado final. A continuación se presentan los primeros pasos realizados hacia el terreno de la captura y tratamiento de magnitudes físicas, así como las conclusiones alcanzadas y los resultados obtenidos tras las pruebas.

F.1. ARQUITECTURA DE LA PLATAFORMA DE PRUEBAS

La plataforma CAEAT y los servicios que la rodean han sido diseñados para ser un software muy versátil que pueda ser ejecutado en multitud de dispositivos distintos. A priori, los únicos requisitos que deberán cumplir dichos dispositivos serán disponer de potencia de procesado suficiente como para ejecutar una Máquina Virtual de *Java* y ser aptos para trabajar sobre una red TCP / IP.

Dado que en la fase actual de desarrollo de la plataforma se desconocen los entornos en los cuales se acabará utilizando CAEAT, se ha querido realizar un testeo de la plataforma en un escenario lo más heterogéneo posible. Para ello se ha definido en primera instancia, y de manera totalmente teórica, una arquitectura tipo que define un posible escenario de trabajo de los servicios desarrollados durante el presente proyecto. Una vez conocidos los recursos hardware y software puestos a disposición de la fase de pruebas, se ha implementado una arquitectura lo más similar posible a la diseñada de manera teórica.

F.1.1. Entorno de despliegue teórico

Se presenta en la página siguiente el esquema de red de la arquitectura teórica diseñada para la fase de pruebas del presente proyecto. Tal y como se detalló en el capítulo 5.5, se pretende separar el procesado llevado a cabo por las agregaciones que se despliegan como servicios en tres grandes capas: adquisición, control y presentación. Es realista pensar que en un entorno real de producción estas tres capas constituirán sub-redes diferentes dentro de la red local que alojará la federación de servicios. Es por ello que el diseño de la plataforma de pruebas contempla la separación de los dispositivos encargados de cada capa en sub-redes independientes.

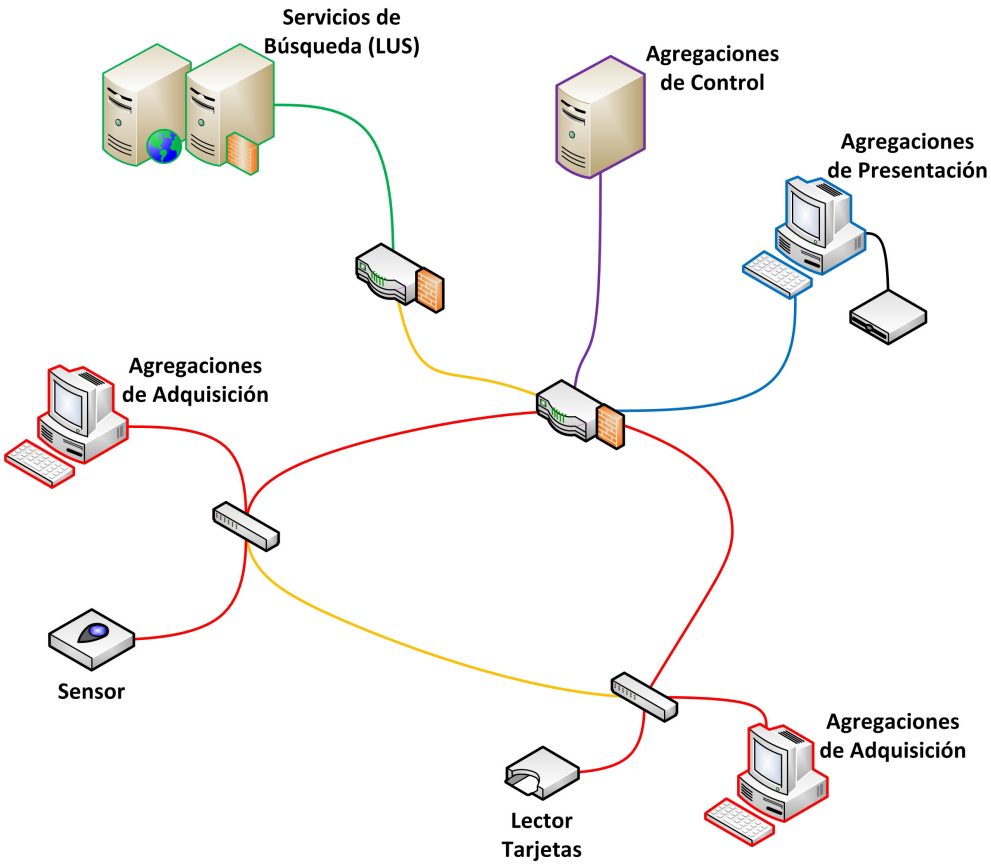

Figura F.1: Arquitectura de red teórica ideada para la fase de pruebas del presente proyecto

- **Adquisición**: representada en el diagrama anterior mediante líneas rojas, la capa de adquisición es la encargada de la interacción con los dispositivos de campo que capturan las magnitudes físicas a monitorizar. En un entorno SCADA, la capa de adquisición estaría formada por todos los PLC's y PAC's encargados de recoger los datos procedentes de los dispositivos de campo, así como del hardware de control que llevaría a cabo su mapeo a objetos del lenguaje *Java* y la inicialización de los mismos como servicios *Jini / Apache River*.

  En este esquema se ha representado un sensor genérico y un lector de tarjetas, cada uno de ellos asociado a un equipo y un *switch* de la sub-red de adquisición. Sin embargo, la interacción de la plataforma CAEAT con dispositivos de adquisición de datos es un campo inexplorado ya que queda fuera del alcance de este proyecto (pese a que se ha realizado un primer paso experimental en esa dirección, tal y como se detalla en este capítulo).

- **Control**: la capa de control estará formada por todos los equipos encargados de alojar las agregaciones de control que (como se detalló en el capítulo 5.5) encapsulan a las de adquisición. En el presente esquema se ha representado un único equipo constituyendo su propia sub-red (en línea de color morado).

- **Presentación**: la capa de presentación está formada por la sub-red que aloja los equipos en los que vivirán las agregaciones de presentación, que encapsularán recursivamente a las de control y adquisición para recibir los datos procedentes de los dispositivos de campo en un formato adecuado y presentarlos al usuario de manera gráfica y amigable. En el esquema mostrado se representa esta red mediante un trazado azul.

  En esta capa se alojarán, si los hubiese, los dispositivos de almacenamiento de datos que servirían como registro histórico de los eventos y singularidades acontecidos durante el transcurso del procesado (véase en el capítulo 9.2.1 la ampliación referente a la interacción con bases de datos SQL).

- **Servidores de *Lookup***: en un escenario de producción real, puede ser una buena práctica la agrupación de los servidores de *Lookup* en una única sub-red. Así se ha representado en el diagrama anterior (en trazado verde), sin embargo conviene no olvidar que la arquitectura *Jini / Apache River* promueve la diversificación de roles, y cualquier máquina de la federación podría perfectamente actuar como servidor LUS. La razón de su agrupamiento en una sub-red radica en facilitar, en el momento de la configuración de la red, la redirección de todos los paquetes *multicast* que realizan búsquedas (*discovery's*) de servidores LUS a la mencionada sub-red, sin necesidad de inundar de paquetes el resto de sub-redes.
- ***Routers*, *gateways* y *firewalls***: en las fronteras de cada sub-red se sitúan los *routers* y *gateways* que se encargarán de redireccionar los paquetes adecuadamente. Habitualmente este tipo de equipamiento hardware cuenta con *firewalls* que no permiten la redirección de ciertos paquetes a menos que se habiliten directrices específicas para ello. Esta fase de pruebas pretende encontrar los requisitos mínimos en términos de configuración de los *firewalls* que son necesarios para el correcto funcionamiento de CAEAT y los servicios desarrollados alrededor de la plataforma.

F.1.2. Recursos hardware y software utilizados

Se presentan en este apartado los recursos hardware y software puestos a disposición de la fase de pruebas del presente proyecto, mediante los cuales se implementará una arquitectura lo más parecida posible a la presentada en el apartado anterior.

F.1.2.1. Equipos de trabajo

Se han empleado un total de cinco máquinas de trabajo pertenecientes a la gama de estaciones de trabajo Dell Optiplex (modelos GX620 y 745). A continuación se detallan las especificaciones técnicas de dichas máquinas:

- **Procesador (Optiplex 745)**: Intel Core 2 Duo 2,40 GHz
- **Procesador (Optiplex GX620)**: Intel Pentium D 3,40 GHz
- **RAM**: 2 GB, DDR2
- **Disco duro**: 160 GB SATA
- **Tarjeta de red**: BroadCom NetXtreme 5754

Figura F.2: Estación de trabajo Dell Optiplex GX620

Uno de los equipos empleados ha sido customizado para disponer de diversas interfaces de red. Como se explica en los apartados subsiguientes, este equipo se reserva para realizar las funciones de *router* y *gateway* dentro de la red.

F.1.2.2. Switch

Se ha empleado un *switch* del modelo 3Com 3CFSU08 para interconectar todos los equipos empleados en la plataforma de pruebas. La gran ventaja que ofrece este *switch* es que es autogestionado y autoconfigurado, siendo capaz de monitorizar los puertos en los que hay dispositivos conectados, ajustar las velocidades de transferencia en consecuencia, conmutar automáticamente entre modos de transmisión MDI y MDIX, etc. El número de puertos ofrecido por este modelo es 8, superior a los 6 que se requieren (4 equipos con una interfaz de red y 1 con dos de ellas), y suficiente como para permitir futuras ampliaciones de la plataforma.

Figura F.3: Switch 3Com 3CFSU08

F.1.2.3. Sistemas operativos

La gama de sistemas operativos utilizados en la plataforma de pruebas es la siguiente:

- ***Windows XP Service Pack 3***: el presente proyecto se ha desarrollado a caballo entre los sistemas operativos *Windows XP* y *Windows* 7, por lo que su correcto funcionamiento sobre estas dos plataformas está garantizado. Pese a ello, se ha preferido que alguna de las dos plataformas contase con presencia en el banco de pruebas.
- ***Ubuntu* 11.10**: a la hora de elegir una distribución del sistema operativo UNIX para tener presencia en el banco de pruebas, se ha optado por *Ubuntu* por ser la que cuenta con más popularidad entre el público general y por ofrecer gran facilidad de utilización y configuración.
- ***Windows Server 2008***: como se detalla en el apartado de conclusiones del presente capítulo, la resolución de nombres de *host* (DNS) en una federación *Jini / Apache River* es un requisito imprescindible para el buen funcionamiento de las mismas. *Zentyal*, el sistema operativo que se presenta a continuación, ofrece la implementación de un servidor DNS. Sin embargo, dado que no se dispuso de *Zentyal* desde el primer momento de funcionamiento del banco de pruebas, se utilizó *Windows Server 2008* a la vez como servidor DNS y como un integrante más de la federación de servicios.
- ***Zentyal* 3.0**: se trata de un sistema operativo basado en UNIX (de código abierto y licencia GNU) específico para actuar como servidor de red. A menudo es visto como una alternativa de código abierto a *Windows Server*, especialmente útil para pequeñas y medianas redes corporativas. Su mayor característica radica en ofrecer funciones de gestión de la infraestructura de red, implementando características como *routing*, *firewall*, filtrado de

paquetes, NAT, redirección de puertos, servidor DNS, servidor DHCP, monitorización y moldeado del tráfico, etc.

En el banco de pruebas diseñado se ha reservado un equipo específico para la ejecución de *Zentyal*. El objetivo que se ha perseguido es el de utilizar dicho equipo como si de un *router* / *gateway* se tratase, aprovechando la facilidad de configuración que *Zentyal* ofrece a través de su interfaz gráfica. De esta forma se podrán establecer de manera sencilla reglas en el *firewall* del sistema, para poder determinar cuáles son necesarias para el correcto funcionamiento de los servicios de CAEAT / *SeNetComponents*. El equipo en el cual se ejecuta *Zentyal* dispone de dos interfaces de red para poder actuar de puente entre dos sub-redes distintas.

La última distribución estable de *Zentyal* es la 3.0 y se puede obtener desde la referencia [**21**], así como consultar sus manuales y documentación oficial.

Figura F.4: Logotipo de la plataforma de gestión de infraestructura de red *Zentyal*

F.1.2.4. Sensores Phidgets

Pese a que la captura y tratamiento de magnitudes físicas quedaba inicialmente fuera del alcance del presente proyecto, se ha decidido dar un primer paso experimental en este campo mediante electrónica de bajo coste que se adaptase bien al entorno propuesto por la plataforma CAEAT y sus servicios. Se han empleado una serie de sensores USB de la compañía *Phidgets* que permiten mapear magnitudes físicas a parámetros fácilmente accesibles mediante un entorno de programación compatible con la inmensa mayoría de lenguajes de programación actuales.

La piedra angular de estos sensores consiste en un elemento hardware alimentado a través del puerto USB que cuenta con las siguientes entradas y salidas:

- **Entradas digitales**: entradas que detectan un cambio de voltaje de 0 a 5 voltios que sirven para la conexión de elementos de interacción humana de naturaleza digital (interruptores, pulsadores, etc.). El estado de estos dispositivos (abierto / cerrado) es accesible programáticamente a través del entorno de programación de *Phidgets*.
- **Salidas digitales**: salidas que se pueden establecer de manera programática a voltaje bajo (0 V) o voltaje alto (5 V). Sirven para la conexión de sencillos elementos hardware de información humana (diodos luminosos, alarmas, etc.).
- **Entradas analógicas**: entradas que son capaces de muestrear una señal de entrada en todo el rango dinámico entre 0 y 5 voltios y mapearla a un valor numérico comprendido entre 0 y 1000, accesible de manera programática a través de la interfaz de programación ofrecida por *Phidgets*. Estas entradas analógicas son las que se han utilizado en las presentes pruebas, ya que sirven para la conexión de sencillos sensores capaces de realizar la medición de magnitudes sencillas tales como la temperatura, la luminosidad, la presión, el movimiento, etc.

Toda la información, documentación y especificaciones de programación relativas a *Phidgets* puede ser consultada en la referencia [**22**].

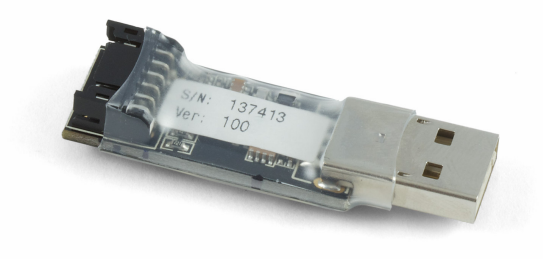

Figura F.5: Modelo de *PhidgetInterface* utilizado durante la fase de pruebas del proyecto

F.1.2.5. Otro hardware

Para el correcto manejo y monitorización de los cinco equipos implicados en la plataforma de pruebas se han empleado dos monitores conectados a sendos *switches* de pantalla. Tres de los equipos han sido conectados a un *switch* de pantalla, mientras que los dos restantes se han conectado al otro. De esta manera es posible visualizar simultáneamente la salida de vídeo de dos de los equipos, pudiendo conmutar rápidamente al resto de ellos mediante los controles de los dos *switches* de pantalla.

F.1.3. Arquitectura implementada

Dado que para desempeñar los papeles de *router* y *gateway* se cuenta con un equipo con dos interfaces de red, se ha optado por simplificar el esquema ideal propuesto al inicio del presente capítulo para implementar únicamente dos sub-redes independientes. Esta topología de red es suficiente para testear las funcionalidades deseadas, y unas pruebas exitosas sobre esta arquitectura garantizan que la introducción de complejidad en la infraestructura de red no acarreará problemas no previstos por las pruebas aquí desempeñadas. La topología final de la red de pruebas se muestra a continuación:

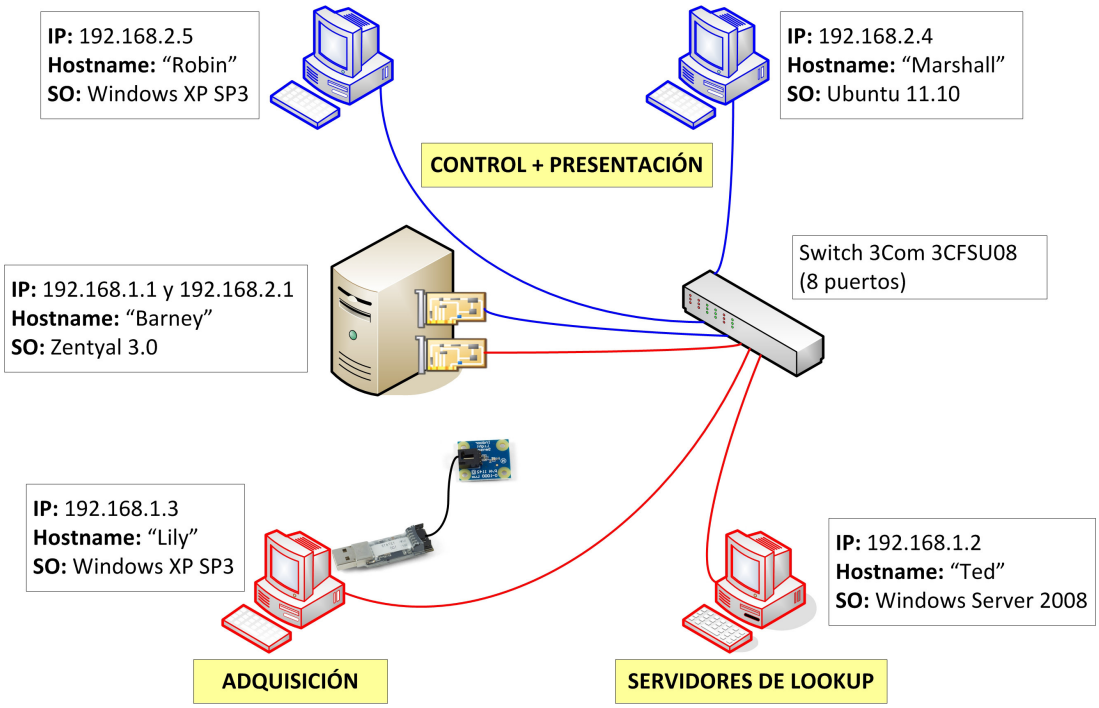

Figura F.6: Topología de red implementada durante la fase de pruebas del proyecto

Puesto que todas las máscaras de red del equipamiento se han establecido a 255.255.255.0, se puede comprobar que la configuración mostrada separa los cuatro equipos de trabajo en dos sub-redes lógicas independientes, con el equipo central (que ejecuta *Zentyal*) actuando como *gateway* entre ellas. *Zentyal* también es el encargado de la ejecución del servidor DNS, que mapea los nombres y las direcciones IP de todos los integrantes de la red.

En una de las redes se ha alojado el equipo que publicará las agregaciones de adquisición. Esto significa que este equipo monitorizará los sensores *Phidgets* que a él se conecten y publicará las agregaciones de componentes específicas que permitan un manejo directo de los datos. En el otro equipo de esta sub-red se han inicializado una serie de servidores de *Lookup* correspondientes a los tres grupos de registro que se pretende implementar ("Adquisición", "Control" y "Presentación"). En la práctica, además, se ha experimentado exitosamente con la ejecución de los servidores LUS repartiendo sus instancias aleatoriamente entre los equipos de la red (incluido el *gateway*).

Por simplicidad, las sub-redes de control y presentación se han fusionado en una sola. Las agregaciones que es posible generar actualmente con CAEAT no disponen por el momento de complejidad suficiente como para que su proceso pueda descomponerse claramente en las tres capas tradicionales de un sistema de supervisión SCADA.

Se han probado con éxito todas las funcionalidades de CAEAT descritas en esta memoria desde una sub-red hacia la otra. Los resultados y conclusiones obtenidos, junto con la lista de requisitos mínimos y limitaciones a nivel de configuración de red que se han detectado, se presentan al final de este capítulo.

F.2. CAPTURA DE MAGNITUDES FÍSICAS

Como se avanzaba en el apartado anterior, se ha hecho uso de los dispositivos de electrónica de bajo coste *Phidgets* para capturar magnitudes del mundo físico e introducirlas en las cadenas de procesado de las agregaciones y servicios generados por CAEAT, para evitar simular estos *inputs* mediante software. En concreto, se ha experimentado con un sensor de vibración, un sensor de presión y un sensor de luminosidad.

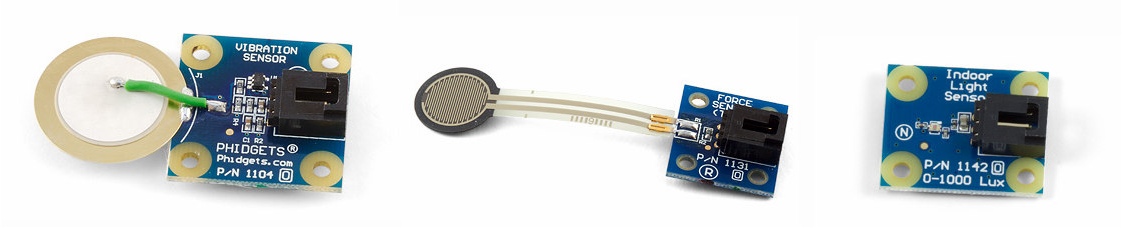

Figura F.7: Sensores *Phidgets* de vibración, presión y luminosidad

Estos sensores son conectados al elemento hardware que convierte las tensiones generadas por los mismos a un valor numérico comprendido entre 0 y 1000 y listo para ser leído a través del puerto USB haciendo uso de los métodos proporcionados por el entorno de programación (las API's) proporcionadas por el fabricante de los sensores. Para cada uno de los dispositivos, el fabricante ofrece en su documentación oficial (véase la referencia [**22**]) una fórmula que permite la conversión del valor numérico ofrecido por la interfaz USB a una unidad del Sistema Internacional de medidas. Es responsabilidad del programador la correcta aplicación de esta formula en el desarrollo del software encargado de la recogida y tratamiento de los datos.

Se ha desarrollado una nueva librería de componentes, llamada *LibreriaPhidgets*, conteniendo un componente por cada uno de los dispositivos empleados. El modelo de programación en el lenguaje *Java* implementado en los sensores *Phidgets* se adapta a la perfección al modelo *JavaBeans* empleado por CAEAT. La clase *Java* encargada del modelado de los sensores es capaz de lanzar eventos de cambio de valor de una propiedad cuando se detecta una variación en el valor del

atributo numérico asociado al sensor. Además, es posible establecer el salto mínimo que se necesitará para que tal lanzamiento de evento se lleve a cabo, ofreciendo así un mecanismo para controlar la sensibilidad de los sensores.

Tomando lo anterior en consideración, un sensor *Phidget* es fácilmente modelable como cualquier otro *bean*, con propiedades a las cuales es posible acceder (algunas serán de sólo lectura, como la salida del sensor), lanzamiento de eventos ante el cambio de dichas propiedades, etc. La particularidad radica en que el *bean* encapsulará la clase *Java* desarrollada por *Phidgets* que permite la comunicación con el dispositivo en cuestión. La siguiente figura muestra las propiedades disponibles en el sensor de presión, cuyo valor de salida es fácilmente convertible a una unidad familiar como el gramo mediante la aplicación de una formula matemática:

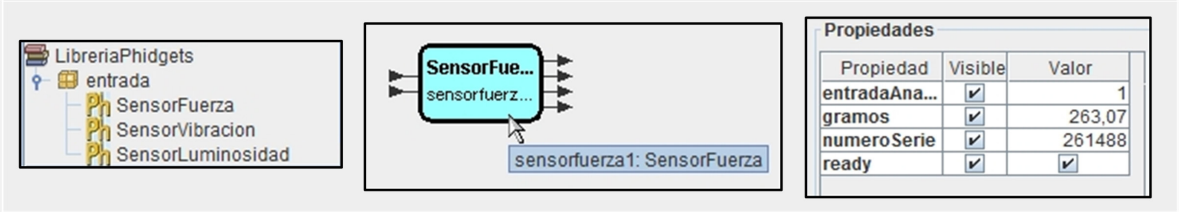

Figura F.8: Representación del sensor de presión (o fuerza) como un componente de CAEAT

El *bean* que modela el sensor tiene como atributos el número de serie de la pieza de hardware encargada de la comunicación USB (requisito imprescindible para poder abrir una comunicación con ella), el número de la entrada analógica en la cual se ha conectado el sensor (el hardware usado únicamente dispone de dos entradas, pero existen productos que cuentan con hasta ocho), un atributo *boolean* que indica si el canal de comunicación con el *Phidget* se ha iniciado correctamente y el valor de lectura, internamente convertido a la unidad correspondiente (gramos en el caso del sensor de presión). En otros sensores, como el de vibración, se ha habilitado un atributo para elegir la sensibilidad del sensor (la diferencia mínima en el valor de salida que se considerará como una vibración detectada).

La figura siguiente ejemplifica el uso de los componentes encargados de la interacción con los sensores y al mismo tiempo muestra la utilidad real de la plataforma CAEAT como sistema de adquisición y supervisión de datos. Se ha exportado y publicado el componente *SensorVibracion* como un servicio. Un usuario distinto, en otra máquina de la federación, ha obtenido el *proxy* hacia dicho servicio y lo ha rodeado de componentes locales que realizan la función de contar el número de vibraciones detectadas y, adicionalmente, hacer saltar una alarma (un atributo *boolean* que se pasa a *true*) en caso de que el número de vibraciones sea superior a 10.

Como se puede observar, adicionalmente el usuario ha dotado a su agregación de componentes gráficos mediante los cuales ha construido una interfaz de usuario a través de la cual puede observar de manera amigable los datos recogidos y las alarmas lanzadas. Cada vez que se detecta una vibración se hace sonar una señal acústica mediante el componente *Beeper* (representado como una campana en la interfaz), mientras que en caso de detectarse un número de vibraciones superior a 10 se activará la alarma correspondiente mediante el paso a rojo del componente LED (representado como una luz de semáforo).

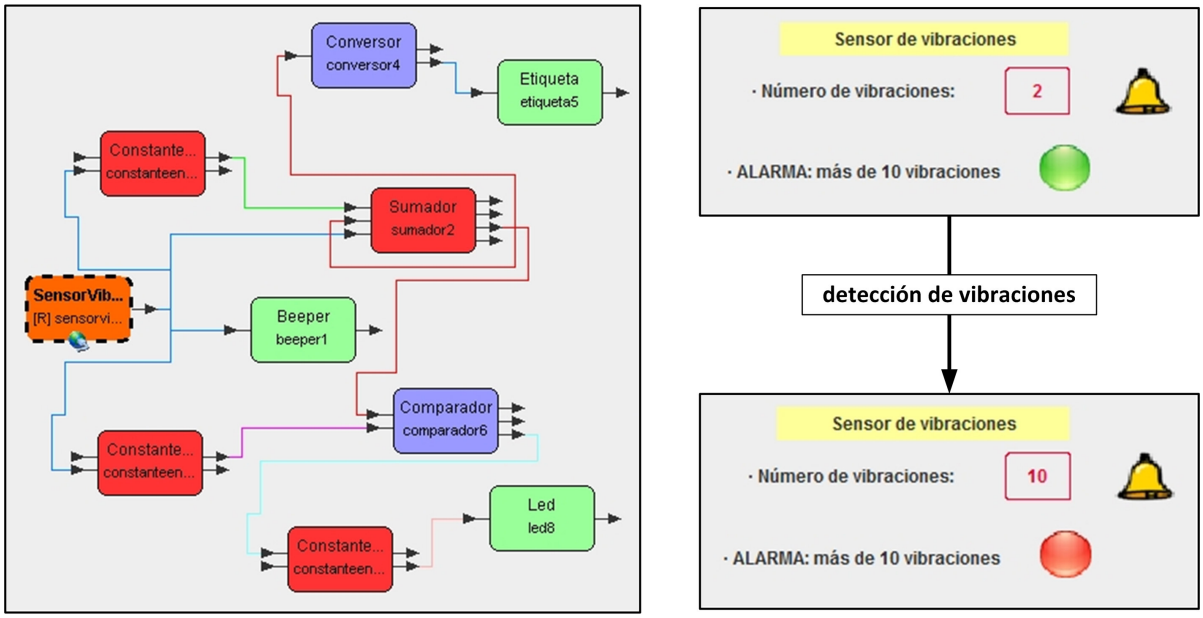

Figura F.9: Agregación empleada para testear el sensor de vibraciones

F.3. CONCLUSIONES Y RESULTADOS

Se presenta a continuación a modo de resultados una lista de los requisitos imprescindibles de configuración necesarios para el correcto funcionamiento de la plataforma CAEAT y sus servicios asociados en un entorno distribuido como el implementado durante esta fase de pruebas:

- Dado que el entorno *Jini / Apache River* emplea el envío de paquetes *multicast* a la red para la búsqueda de servicios, se debe garantizar que los nodos intermedios de la red son capaces de redireccionar adecuadamente este tipo de tráfico. Pese a que la mayoría de *routers* comerciales actuales implementan por defecto esta característica, habitualmente en las redes corporativas el tráfico *multicast* está limitado o restringido.

  Se debe garantizar que el tráfico *multicast* esté habilitado en todos los nodos intermedios (*routers*) de la red, así como su aceptación por parte de todos los equipos pertenecientes a la federación (el proceso para habilitar la respuesta a los paquetes *multicast* varía de un sistema operativo a otro).

- La dirección IP *multicast* empleada por las API's de *Jini / Apache River* para llevar a cabo el *discovery* o búsqueda de servicios es la 224.0.1.84. Los paquetes se envían al puerto UDP 4160, oficialmente establecido como el puerto reservado para el protocolo de *discovery* de *Jini*. Es requisito imprescindible que los paquetes dirigidos dicha dirección IP no resulten bloqueados por ningún *firewall* intermedio ni de los equipos de trabajo. También se debe garantizar, para cada uno de los equipos pertenecientes a la federación, que los paquetes dirigidos al puerto UDP 4160 no sean descartados por las reglas de seguridad de los *firewalls*.

- El entorno *Jini / Apache River* y los programas y servicios que lo rodean (como por ejemplo los servidores de *Lookup*) asumen que existe en la red un sistema de resolución de nombres de dominio y *hosts* (DNS) activo y operando correctamente. Las API's hacen un uso indistinto de los nombres de *host* y las direcciones IP de los equipos de la federación en las diferentes funciones de búsqueda e intercambio de objetos, de manera que se hace necesaria la instalación y permanente ejecución de un servidor DNS en la red formada por las entidades de la federación *Jini / Apache River*.

En caso de existir un *gateway* (como es el caso del banco de pruebas aquí presentado, en el que el equipo que ejecuta *Zentyal* actúa de *gateway* entre dos sub-redes) el servidor DNS se ejecutará por lo general en él. Sin embargo, en infraestructuras de red grandes también sería habitual disponer de un servidor DNS en cada sub-red.

- Tal y como se ha comentado anteriormente y comprobado experimentalmente, la visibilidad de los paquetes *multicast* se limita, por defecto, a la misma sub-red dentro de la cual han sido generados. Los servidores LUS alojados en una sub-red distinta a la cual pertenece la entidad que realiza la búsqueda no son visibles para ésta. Con tal de evitar esta situación y dotar de visibilidad total a las entidades pertenecientes a la federación de servicios se debe configurar adecuadamente el redireccionado de los mensajes *multicast* en el *gateway* encargado de enlazar las distintas sub-redes.

  En el caso del banco de pruebas diseñado, dicha responsabilidad recae sobre la máquina que ejecuta *Zentyal* y que actúa como *gateway* entre las dos sub-redes desplegadas. *Zentyal* permite configurar reglas de enrutamiento de paquetes de una manera rápida y mediante una interfaz gráfica. Dado que todo paquete cuyo puerto de destinación sea el UDP 4160 será necesariamente un paquete *multicast* que trata de realizar un *discovery*, bastará con establecer reglas en *Zentyal* para que la recepción de estos paquetes por una de las interfaces de red suponga su redirección automática al resto de interfaces de red. De esta forma el paquete multicast "saltará" el *gateway* de manera transparente y alcanzará la totalidad de las sub-redes desplegadas.

- Algunas de las funcionalidades desarrolladas para CAEAT requieren el uso de un servidor FTP (descarga y subida de archivos de librerías, subida de imágenes, etc.). Ello implica que dicho protocolo no se deba encontrar limitado en la red de servicios. Todos los dispositivos y nodos intermedios de la red deben ser aptos para manejar tráfico FTP y en ningún caso se deben descartar los paquetes dirigidos al puerto TCP 21 (puerto de destino de los paquetes pertenecientes a una sesión FTP).

- Como se ha explicado en los capítulos correspondientes, la exportación e inicialización de servicios *Jini / Apache River* requiere de la utilización de un servidor HTTP. El puerto en el cual opera dicho servidor es indistinto y es posible configurar CAEAT para adaptar su procesado al puerto escogido; sin embargo, a nivel de infraestructura de red, resulta imprescindible que los paquetes dirigidos a dicho puerto no sean descartados por ningún nodo intermedio ni por ninguno de los equipos pertenecientes a la federación.

F.3.1. Ampliaciones de la fase de pruebas

Durante esta primera fase de testeo realizada a la plataforma CAEAT y a sus servicios se han llevado a cabo pruebas sobre diferentes sistemas operativos y utilizando configuraciones de red muy diversas gracias a las posibilidades ofrecidas por *Zentyal*. Sin embargo, la plataforma de pruebas puede ser ampliable para constituir un escenario aún más heterogéneo. De la misma manera en que se dejan líneas de trabajo abiertas en referencia al desarrollo de la plataforma CAEAT, también se listan a continuación posibles ampliaciones que se pueden implementar sobre un banco de pruebas como el construido (o desplegar sobre una plataforma de pruebas completamente nueva).

- Las librerías de *Jini / Apache River* realizan por defecto la búsqueda de servicios en la red (*discovery*) mediante paquetes *multicast* enviados a través de todas las interfaces de red disponibles en la máquina. En el presente banco de pruebas no se ha comprobado qué ocurre si una máquina dispone de una interfaz de red inalámbrica.

  Resultaría interesante añadir al banco de pruebas una nueva sub-red únicamente formada por dispositivos *wireless* (por ejemplo, ordenadores portátiles) y comprobar si estas

interfaces, correctamente configuradas, son visibles en la federación de servicios *Jini / Apache River* o si es necesaria algún tipo de configuración adicional de las funciones de búsqueda.

- Pese a que prácticamente todas las máquinas pertenecientes al banco de pruebas disponen de un sistema operativo distinto, todos ellos pertenecen a las dos mismas grandes familias (*Windows* y UNIX). Sería de gran utilidad testear el rendimiento de CAEAT y sus servicios en sistemas operativos completamente distintos, siendo la primera y más inmediata elección el popular sistema *MacOS* usado por los equipos desarrollados por *Apple*.
- El banco de pruebas podría ser progresivamente ampliado para incluir más sub-redes y de esta manera asemejarse más a la arquitectura de pruebas ideal que se diseñó en el primer apartado del presente capítulo. La agregación de un *router* con funciones de *gateway* permitiría establecer una cantidad superior de sub-redes. Si al mismo tiempo se provoca que todo el tráfico pase a través del equipo en el cual se ejecuta *Zentyal*, la monitorización de los paquetes y su filtrado mediante interfaz gráfica seguirá siendo posible.
- Este banco de pruebas ha realizado una primera experimentación con la captura y el tratamiento de datos procedentes de magnitudes del mundo físico, gracias a los sensores electrónicos USB de bajo coste *Phidgets*. Sin embargo, progresivamente y a medida que la plataforma crezca se hará necesaria la experimentación con dispositivos más profesionales. Sería conveniente la utilización de electrónica que implementase alguno de los protocolos estándar que se desea hacer compatibles con la plataforma CAEAT (véase la lista de líneas de trabajo abiertas en el capítulo 9.2.1).
- Por defecto, el protocolo de *discovery* utilizado por las API's de *Jini / Apache River* para realizar la búsqueda de servicios emplea mensajes *multicast* para la comunicación con todas las entidades de la federación. Como se ha comprobado en el banco de pruebas, el alcance de estos mensajes es limitado. Sin embargo, las API's también soportan una versión *unicast* del protocolo, en el cual se utilizan paquetes TCP tradicionales "atacando" directamente la dirección IP en la cual reside el servidor de *Lookup*.

  Esta característica viola la filosofía de la Arquitectura Orientada a Servicios, ya que se debe conocer de antemano la ubicación del servidor LUS y únicamente se lleva a cabo la comunicación con uno de ellos, no con la "nube" de entidades que pueblan la federación. Sin embargo, teóricamente esta característica permite la realización de *discovery's* a través de Internet. Resultaría interesante experimentar con esta funcionalidad para averiguar si es posible la obtención de servicios a través de Internet, determinando al mismo tiempo los requisitos y limitaciones de la operativa necesaria para llevarla a cabo.

G. HERRAMIENTAS SOFTWARE UTILIZADAS

Este anexo presenta las herramientas software utilizadas durante la elaboración del presente proyecto, junto con una breve descripción de sus usos y características y las razones por las cuales se ha elegido su utilización. No se incluyen aquí menciones al servidor HTTP y al servidor de *Lookup Reggie* (ambos incluidos en la distribución de *Apache River*) ni al servidor *Apache FTP Server*, cuyos usos y características han sido tratados en capítulos anteriores de esta memoria.

G.1. ECLIPSE

Eclipse es un entorno de desarrollo software ideado principalmente para la creación de aplicaciones en el lenguaje de programación *Java*. Proporciona un IDE (entorno de desarrollo integrado) y sus funcionalidades son ampliables mediante la instalación de *plugins*. Junto con *NetBeans*, el entorno de desarrollo *Eclipse* es el más utilizado en la actualidad para la programación en el lenguaje *Java*.

La plataforma *Eclipse* está creada y respaldada por la *Eclipse Foundation*, una organización sin ánimo de lucro fundada en 2003 como comunidad de desarrollo de proyectos de código abierto. El entorno de desarrollo software *Eclipse* representa su proyecto más notorio, siendo el objetivo de la fundación la "construcción de plataformas de desarrollo abierto formadas por *frameworks* extensibles, herramientas y entornos de ejecución capaces de construir, desplegar y mantener software a lo largo de todo su ciclo de vida". Algunas de las entidades privadas más notorias que tienen representación en la *Eclipse Foundation* son *Oracle*, *Nokia*, IBM o SAP. El portal de la fundación *Eclipse*, con información sobre todos sus proyectos, puede ser consultado en la referencia [**17**]. Desde dicho portal es posible obtener también las versiones del entorno de programación para las distintas plataformas existentes.

Figura G.1: Logotipo de la plataforma de desarrollo software *Eclipse*

Entre las características y ventajas que han llevado a la elección de este entorno de desarrollo para su utilización en el presente proyecto se encuentran las siguientes:

- Se trata de una plataforma de código abierto ampliamente extendida y fuertemente respaldada por la *Eclipse Foundation*. Es previsible que este apoyo y soporte sea extensible durante largo tiempo en el futuro.
- Es multiplataforma y se distribuyen versiones para la totalidad de arquitecturas y sistemas operativos existentes en la actualidad.
- Pese a ser conocida principalmente como un IDE de desarrollo de aplicaciones *Java*, *Eclipse* es una plataforma capaz de realizar otras muchas funciones gracias a su ampliación por medio de la instalación de *plugins*.
- Gracias al uso de *plugins* que se acaba de mencionar, *Eclipse* puede ser ampliado para operar como IDE en otros lenguajes de programación que incluyen C, C++, COBOL, *Fortran*, *Perl*, PHP, *Python* o *Ruby*.

G.2. SUBVERSION

Subversion (coloquialmente abreviado SVN) es un software de control de versiones y revisiones de código abierto y distribuido bajo la licencia *Apache*. En el contexto del desarrollo de un proyecto, un sistema de control de versiones como *Subversion* se utiliza para mantener y controlar la versión actual de los archivos que forman dicho proyecto, pudiendo realizar copias o *tags* del proyecto en cualquier momento, y permitiendo revertir cambios para volver a versiones anteriores de parte o la totalidad del trabajo.

Pese a ser ampliamente utilizado en proyectos de desarrollo software, *Subversion* se puede utilizar para llevar un control de las versiones y los cambios realizados sobre cualquier tipo de fichero: páginas web, documentos de texto, hojas de cálculo, etc. *Subversion* surge en el año 2000 como una evolución de CVS (*Concurrent Versions System*), la plataforma de gestión de versiones de código abierto previamente existente, lanzada en 1990.

Figura G.2: Logotipo del sistema de control y revisión de versiones *Subversion*

Subversion almacena la versión actual del proyecto junto con todo su histórico en un servidor central o repositorio. Los clientes que trabajan en el proyecto obtienen una copia de la versión actual del trabajo en una operación llamada *check-out*. Tras trabajar con la copia local, los clientes deben retribuir los cambios a la versión actual del repositorio mediante una operación de subida, denominada *check-in* o *commit*.

Un simple cambio realizado en la versión actual del repositorio, pese a afectar únicamente a un archivo, se refleja en el incremento del índice que indica la revisión actual del repositorio alojado en el servidor. De esta manera todos los clientes son conscientes al instante de que existen diferencias entre el entorno de trabajo local y la versión actual del repositorio. Los clientes pueden obtener la última revisión alojada en el servidor mediante una operación de *update*.

El índice que indica el estado del repositorio en un momento determinado sirve como *snapshot* del proyecto, pudiendo revertir su estado a cualquiera de los índices anteriores en cualquier momento. El índice de trabajo actual se denomina *trunk*. A partir de un índice determinado también se pueden generar *tags* y *branches*:

- ***Tag***: versión significativa del proyecto que normalmente se genera tras alcanzar un hito importante y que se guarda en el repositorio como copia de seguridad.
- ***Branch***: ramificación del proyecto que normalmente avanza de manera concurrente a la corriente de trabajo principal o *trunk*. Una ramificación puede acabar constituyendo un *spin-off* del proyecto (se debería generar un nuevo repositorio para el nuevo proyecto), puede acabar discontinuada (abandonándose dicha ramificación) o puede acabar siendo fusionada nuevamente con la corriente de trabajo principal o *trunk*.
- ***Trunk***: versión de trabajo actual del proyecto.

La plataforma SVN también sirve para controlar la modificación concurrente de los ficheros. Para ello, el servidor únicamente acepta cambios realizados sobre la última versión de dicho fichero, razón por la cual los usuarios deben realizar operaciones de *update* con frecuencia. Es posible, además, el uso de una funcionalidad mediante la cual un usuario puede marcar un fichero como propio y

bloquearlo, evitando que otros usuarios trabajen sobre él hasta que no se libere y se entreguen los cambios al repositorio.

Pese a las medidas de protección, es posible que surjan conflictos de edición entre las versiones generadas por dos usuarios distintos. En ese caso, *Subversion* implementa el mecanismo de *merge* o fusión de dos ficheros en uno solo. Si esta operación no se puede llevar a cabo automáticamente (por ejemplo, porque implica tomar decisiones sobre qué fragmentos del fichero eliminar) se pide al usuario que resuelva los conflictos manualmente.

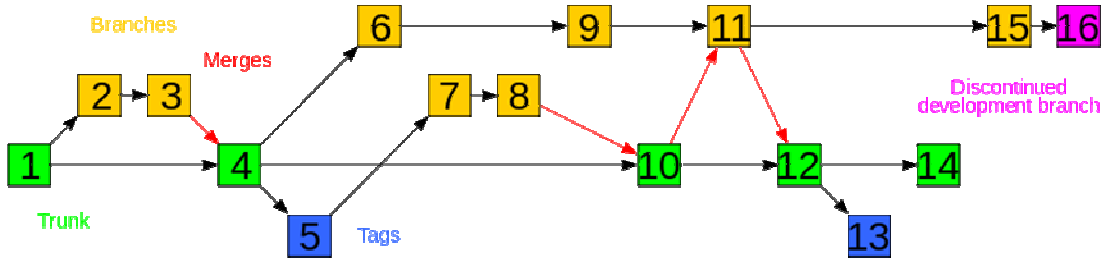

Figura G.3: Esquema de ramificaciones de un repositorio de *Subversion*

Para la utilización de *Subversion* como cliente es necesario el uso de un paquete de software capaz de interactuar con el repositorio. Existen multitud de clientes de SVN, siendo uno de los más conocidos *Tortoise*, para el sistema operativo *Windows*. Sin embargo, la propia plataforma *Eclipse* ofrece como *plugin* un módulo de software llamado *Subclipse* capaz de obtener los archivos de código fuente desde un repositorio SVN, integrando las operaciones de interacción con el repositorio que se acaban de describir en la interfaz gráfica de *Eclipse* de una forma intuitiva y sencilla. Este proyecto ha optado por esta última vía.

En el marco del presente proyecto, la versión inicial del software se encontraba alojada en un repositorio SVN en un servidor. Se ha utilizado *Eclipse* y *Subclipse* para acceder a los recursos del repositorio y obtener el código fuente del proyecto. En un entorno de trabajo mono-usuario, se ha utilizado *Subversion* para mantener en el repositorio una imagen actualizada del proyecto, realizando copias de seguridad (*tags*) con regularidad para poder revertir fácilmente a ellas en caso de error o desviación en el trabajo.

Para más información acerca de *Subversion*, consúltese el portal de *Apache* dedicado a dicho proyecto en la referencia [**18**].

G.3. WIRESHARK

Wireshark es un software gratuito y de código abierto para el análisis de los paquetes transmitidos en una red. Habitualmente se utiliza para detectar problemas en la red, como apoyo durante un proyecto de desarrollo software, o con fines educativos. *Wireshark* es multi-plataforma, ya que utiliza la API *pcap* (*packet capture*) para la captura y el análisis de los paquetes, y se adapta su interfaz de usuario dependiendo del sistema operativo y la arquitectura para la cual se distribuya.

Figura G.4: Logotipo de la plataforma de captura y análisis de paquetes *Wireshark*

Wireshark es compatible con la práctica totalidad de los protocolos de red existentes en la actualidad, por lo que es capaz de diseccionar los paquetes pertenecientes a los distintos protocolos y presentar los campos de información encapsulados en ellos de una manera claramente diferenciada y amigable para el usuario. Un ejemplo de dicha interfaz se muestra a continuación:

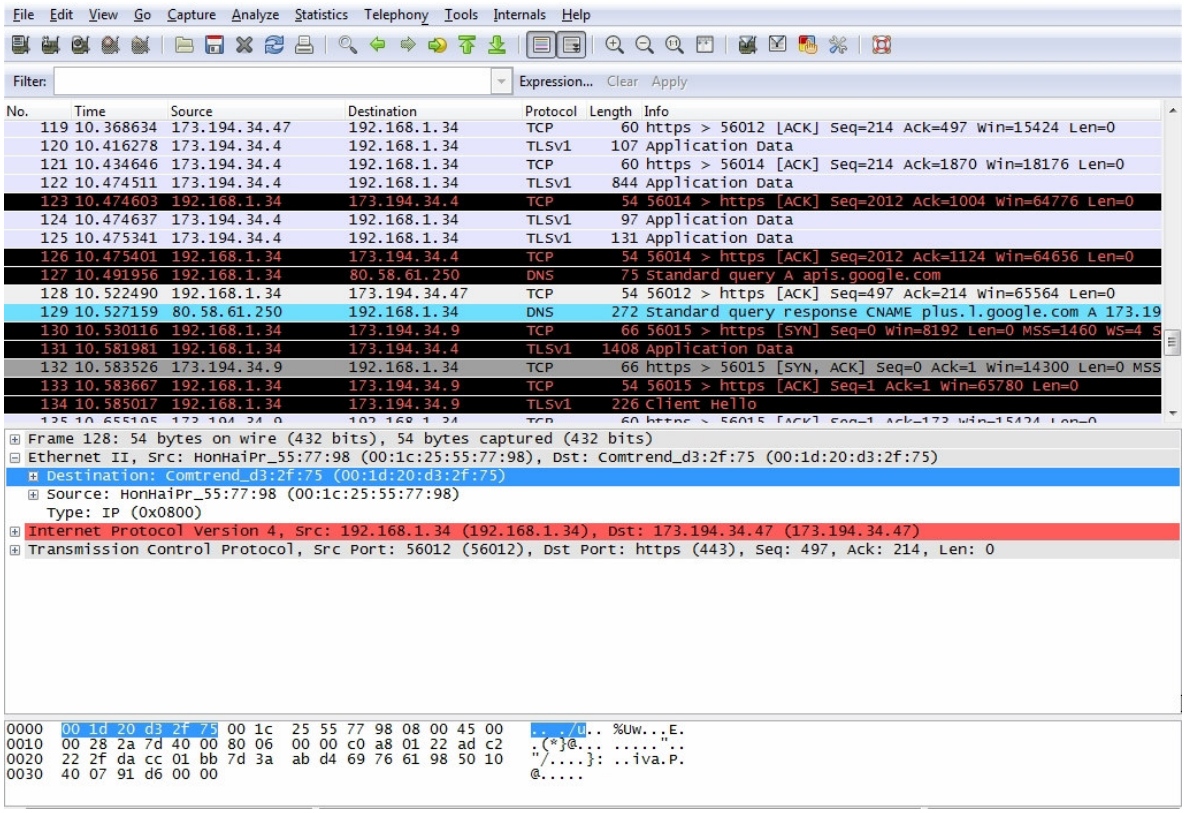

Figura G.5: Interfaz de captura y análisis de paquetes de *Wireshark*

Entre las potentes características y capacidades para el análisis del tráfico de red que ofrece *Wireshark*, cabe destacar las siguientes:

- Los datos pueden ser capturados “on the wire”, es decir, en tiempo real mientras se está llevando a cabo la comunicación.
- Las capturas realizadas pueden ser almacenadas en un fichero para su análisis posterior.
- Es posible capturar tráfico de cualquiera de las interfaces de red disponibles en la máquina.
- Los paquetes capturados pueden ser filtrados de manera que únicamente se muestren al usuario los que siguen un cierto patrón. Es posible establecer filtros atendiendo a multitud de parámetros (protocolo, direcciones IP de origen y destino, puertos, etc.).
- Permite la instalación de *plugins* para la comprensión y análisis de protocolos desconocidos.

En el contexto del presente proyecto, *Wireshark* ha sido utilizado durante la fase de pruebas en una plataforma distribuida real (ver anexo F). Se han capturado los paquetes enviados y recibidos por la interfaz *Ethernet* de los distintos equipos para garantizar la correcta recepción de los mensajes multicast necesarios en el entorno *Jini / Apache River*. Se han analizado los paquetes para corroborar que los destinatarios, su formato, las informaciones encapsuladas en ellos, etc. coinciden con lo esperable según lo estipulado en las especificaciones de la plataforma *Jini*. Los resultados de la fase de pruebas fueron presentados en el anexo F de esta memoria.

G.4. OTROS RECURSOS SOFTWARE

Este apartado hace mención de otros recursos software externos empleados que no constituyen aplicaciones software por sí mismos, como librerías y recursos de código abierto que se han incorporado a la distribución de CAEAT durante la elaboración del presente proyecto.

Para todas las librerías externas incluidas en el proyecto, se ha procurado que la licencia de distribución corresponda a una de las tres licencias software que permiten la reutilización, modificación y distribución del código dentro de aplicaciones que pueden ser licenciadas como software propietario (y por consiguiente, con finalidades comerciales). Estos tres tipos de licencia son los siguientes:

- ***Apache Software License***: fue ampliamente descrita en el capítulo 1.3.1 de esta memoria
- ***Berkeley Software Distribution***: junto con la *Apache License*, se trata de la licencia de distribución menos restrictiva que existe.
- **GNU *Lesser General Public License***: licencia menos restrictiva que GPL puesto que no obliga a que las aplicaciones que hagan uso de ella sean distribuidas también bajo GPL.

G.4.1. Widgets gráficos

A la hora de desarrollar una aplicación con interfaz gráfica, resulta imprescindible para el éxito de dicha aplicación que la interfaz de usuario sea atractiva, cómoda e intuitiva. Para ello, es habitual el uso de librerías externas para dotar a la herramienta de *widgets* complejos a la vez que atractivos que sirvan para la introducción de datos o su visualización por parte del usuario.

El presente proyecto ha tenido la necesidad de implementar un control de selección de fechas, para poder escoger las fechas de caducidad de los servicios en el momento de su publicación. Un control de este tipo no está incluido en los controles por defecto incluidos en el paquete *Swing* de *Java*, por lo que se ha optado por la búsqueda de librerías de código abierto que implementen controles de este tipo usando los elementos comunes de *Swing* (botones, campos de texto, etc.).

La librería externa utilizada ha sido *JCalendar*, que implementa diferentes interfaces de usuario para la selección de fechas.

- **Obtención y documentación**: http://www.toedter.com/en/jcalendar/index.html
- **Licencia**: GNU LGPL (*Lesser General Public License*)
- **Versión utilizada**: 1.4

La librería ofrece diferentes *widgets* de introducción de fechas, altamente configurables tanto a nivel lógico (por ejemplo, es posible mostrar un calendario en formato tradicional o anglosajón) como visual (es posible mostrar el calendario mediante botones, cuadros de texto rotativos, etc.). El resultado de la aplicación de esta librería al presente proyecto ha sido mostrado en capítulos anteriores, por lo que la siguiente captura muestra las interfaces alternativas de introducción de fechas que se pueden conseguir mediante la configuración adecuada de los *widgets* ofrecidos:

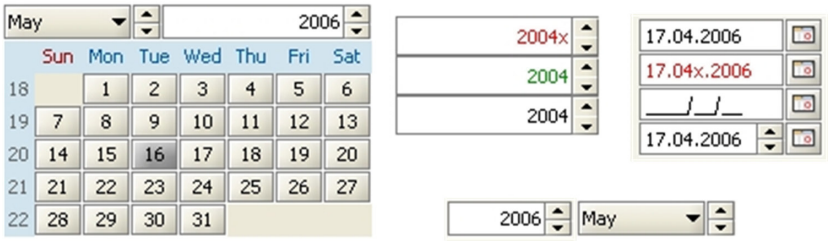

Figura G.6: Diferentes interfaces de introducción de fechas ofrecidas por *JCalendar*

G.4.2. Manejo de fechas

Una vez obtenidas las fechas pertinentes, este proyecto ha tenido la necesidad de realizar operaciones de conversión con ellas: calcular el número de meses, días y años entre dos fechas dadas en formato absoluto, sumar cierta cantidad de tiempo a una fecha, etc. Por lo general, realizar este tipo de operaciones con la distribución actual del entorno de programación *Java* (versión 1.7) resulta posible, pero engorroso.

Las API's de *Java* representan las fechas como un valor absoluto en milisegundos transcurridos desde un origen de tiempos que se sitúa en el día 1 de enero de 1970. Realizar cálculos como los descritos anteriormente (meses, semanas, días, etc.) con dos valores absolutos implica que el programador debe realizar la tarea de escribir manualmente funciones reutilizables capaces de calcular los valores relativos deseados a partir de los absolutos en milisegundos.

Existe no obstante la librería de libre distribución *Joda Time*, creada como un complemento y mejora a las clases de manejo del tiempo proporcionadas por la plataforma *Java*, que no permiten realizar este tipo de cálculos de manera directa. La librería es un proyecto de código abierto que proporciona clases útiles para la realización de todo tipo de cálculos y conversiones entre unidades de tiempo.

- **Obtención y documentación**: http://joda-time.sourceforge.net/
- **Licencia**: *Apache License*
- **Versión utilizada**: 2.1

G.4.3. Look and Feel's externos

El anexo D expuso la inclusión en la plataforma de *Look and Feel's* desarrollados por terceros y seleccionables desde el diálogo de opciones de CAEAT. Este apartado lista las librerías externas que contienen dichos LAF's y que se han añadido al proyecto para dotarlo de esta funcionalidad. Para cada librería, se cita su sitio oficial de obtención, su licencia de distribución y la versión incorporada a la plataforma CAEAT.

G.4.3.1. Liquid Look and Feel

- **Obtención y documentación**: http://sourceforge.net/projects/liquidlnf/
- **Licencia**: GNU LGPL (*Lesser General Public License*)
- **Versión utilizada**: 0.2.9

G.4.3.2. Nimrod Look and Feel

- **Obtención y documentación**: http://personales.ya.com/nimrod/
- **Licencia**: GNU LGPL (*Lesser General Public License*)
- **Versión utilizada**: 1.2

G.4.3.3. Squareness Look and Feel

- **Obtención y documentación**: http://squareness.beeger.net/
- **Licencia**: BSD (*Berkeley Software Distribution*)
- **Versión utilizada**: 2.0

G.4.3.4. Tiny Look and Feel

- **Obtención y documentación**: http://www.muntjak.de/hans/java/tinylaf/index.html
- **Licencia**: GNU LGPL (*Lesser General Public License*)
- **Versión utilizada**: 1.4.0

G.4.3.5.Tonic Look and Feel

- **Obtención y documentación**: http://digitprop.com/tonic-lf/
- **Licencia**: GNU LGPL (*Lesser General Public License*)
- **Versión utilizada**: 1.0.6

G.4.3.6. JTattoo

- **Obtención y documentación**: http://www.jtattoo.net/
- **Licencia**: BSD (*Berkeley Software Distribution*) con restricciones
- **Versión utilizada**: 1.6.3

La librería *JTattoo*, que contiene una gran cantidad de *Look and Feel's* compatibles con *Swing*, se distribuye bajo una licencia de código abierto siempre y cuando sea usada para el desarrollo de aplicaciones de finalidades no-comerciales. Si se desea comercializar la aplicación y licenciarla como propietaria, se debe adquirir una licencia a los creadores de la librería.

En el momento de finalización del presente proyecto, se ha decidido mantener la librería JTattoo como parte integrante de la plataforma CAEAT con finalidades de debugado y pruebas. Sin embargo, cuando en el futuro se licencie la plataforma como software con finalidades comerciales, el paquete *JTattoo* deberá ser eliminado de la distribución a menos que haya cambiado su licencia con el transcurso del tiempo.

REFERENCIAS

[**1**] http://river.apache.org/doc/spec-index.html

Especificaciones completas de los diferentes protocolos y funcionalidades implementados por la tecnología Jini.

[**2**] http://docs.oracle.com/javase/1.5.0/docs/guide/serialization/spec/serialTOC.html

Especificaciones oficiales de las API's del lenguaje de programación Java que tratan con la serialización de objetos para su almacenaje y/o transmisión por la red.

[**3**] http://river.apache.org

Sitio oficial de Apache River. En él se puede obtener la última versión de la plataforma, así como consultar la documentación oficial y las especificaciones de las API's. También ofrece numerosos manuales y tutoriales de iniciación y una lista de distribución de correo para poner en contacto a usuarios y desarrolladores.

[**4**] http://www.apache.org/licenses/LICENSE-2.0

Definición, términos y condiciones de la versión 2.0 de la licencia de distribución de software libre Apache License.

[**5**] http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

Especificaciones oficiales del estándar JavaBeans creado por Sun Microsystems.

[**6**] http://www.jdom.org/

Sitio oficial de la librería JDOM, entorno de trabajo para la manipulación de documentos XML de manera sencilla en lenguaje de programación Java.

[**7**] http://docs.oracle.com/javase/1.5.0/docs/guide/jar/jar.html

Especificaciones oficiales del formato de los archivos jar. Incluye información acerca de la semántica a seguir a la hora de escribir un archivo de manifiesto comprensible por la Máquina Virtual de Java.

[**8**] http://river.apache.org/doc/api/com/sun/jini/reggie/package-summary.html

Especificaciones y posibilidades de configuración de Reggie, la implementación de servidor de Lookup usada en el presente proyecto.

[**9**] http://mina.apache.org/ftpserver/

Sitio oficial del servidor FTP "Apache FTP Server", utilizado durante la elaboración de este proyecto.

[**10**] Herrero Andrés, C.: Herramienta *de Edición y Ensamblaje de Componentes y Agregaciones.* Facultat d'Informàtica de Barcelona, 2011

Memoria descriptiva del proyecto de final de carrera del cual el presente proyecto es la continuación natural.

[**11**] Arnold, K.: *The Jini Specification*. Addison Wesley, 1999

Especificaciones de la plataforma Jini ampliadas y comentadas.

[**12**] Edwards, W.K.: *Core Jini*. Prentice Hall, 1999

Introducción a la tecnología Jini desde un punto de vista práctico, mediante el uso de ejemplos ejecutables

[**13**] Christopher, T.W.; Thiruvathukal, G.K.: High-performance Java platform computing. Prentice Hall, 2000

Introducción a la programación multithreading en la plataforma Java. Cuestiones comunes sobre concurrencia y optimización del rendimiento en aplicaciones distribuidas.

[**14**] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, 1995

Definición y explicación de los principales patrones de diseño tradicionalmente empleados en la programación orientada objetos, independientemente del lenguaje.

[**15**] http://www.w3.org/TR/WCAG10/

Recomendaciones del World Wide Web Consortium sobre accesibilidad en el diseño de páginas web.

[**16**] http://www.udc.es/fcs/es/web-to/terapia/asignaturas/toyafam/08tema/UNE139802-

2003.pdf

UNE 139802-2003: *Software. Requisitos de accesibilidad al ordenador. Aplicaciones informáticas para personas con discapacidad*. AENOR, 2003

Normativa española sobre la accesibilidad de las plataformas software.

[**17**] http://www.eclipse.org/

Portal de la Eclipse Foundation, con información acerca de sus miembros, sus eventos y sus proyectos actualmente activos. Desde el portal es posible obtener la última versión del entorno de desarrollo Eclipse para cualquiera de las plataformas existentes.

[**18**] http://subversion.apache.org/

Sitio oficial del proyecto Suvbersion. Ofrece la documentación oficial, soporte y la descarga de la última versión del proyecto.

[**19**] http://www.mysql.com/

Sitio oficial de la plataforma MySQL, que incluye los paquetes descargables para todas las plataformas existentes, documentación oficial, especificaciones, manuales, etc.

[**20**] http://www.hibernate.org/

Sitio oficial de las librerías Hibernate, que ofrece su descarga y su documentación oficial.

[**21**] http://www.zentyal.com/es/

Sitio oficial del sistema operativo basado en UNIX de gestión de infraestructura de red Zentyal, desde el cual es posible obtener su última distribución y consultar manuales y documentación oficial.

[**22**] http://www.phidgets.com/

Sitio oficial de los productos electrónicos de bajo coste Phidgets. Desde el portal es posible la adquisición de los distintos productos del catálogo, la consulta de sus especificaciones y manuales de programación, la obtención de los drivers y las librerías necesarios para su funcionamiento y extractos de código a modo de ejemplos de utilización.